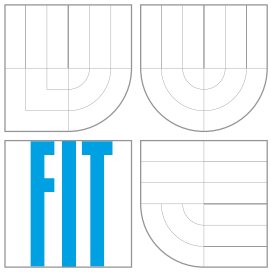
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

FAKTORIZACE CELÝCH ČÍSEL NA GPU

INTEGER FACTORIZATION ON THE GPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ PODHORSKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. IVAN HOMOLIAK

BRNO 2014

Abstrakt

Tato práce pojednává o faktorizaci, tedy rozkladu složených čísel na prvočísla a možnostech její paralelizace. Dále shrnuje nejznámější algoritmy pro faktorizaci a nejznámější platformy pro implementaci těchto algoritmů na grafické kartě. Hlavní část práce se zaobírá návrhem a implementací hardwarové akcelerace současného nejrychlejšího algoritmu na grafické kartě s využitím frameworku OpenCL. Následně je v práci uvedeno srovnání rychlostí akcelerovaného algoritmu implementovaného v rámci této práce s ostatními nejznámějšími verzemi algoritmů pro faktorizaci, zpracovávané sériově. Na závěr je v práci diskutována délka klíče algoritmu RSA potřebná pro bezpečný provoz bez možnosti jejího prolomení v reálném časovém intervalu.

Abstract

This work deals with factorization, a decomposition of composite numbers on prime numbers and possibilities of its parallelization. It summarizes also the best known algorithms for factoring and most popular platforms for the implementation of these algorithms on the graphics card. The main part of the thesis deals with the design and implementation of hardware acceleration current fastest algorithm on the graphics card by using the OpenCL framework. Subsequently, the work provides a comparison of speeds accelerated algorithm implemented in this work with other versions of the best known algorithms for factoring, processed serially. In conclusion, the work discussed length of RSA key needed for safe operation without the possibility of breaking in real time interval.

Klíčová slova

CUDA, OpenCL, celočíselná faktorizace, Kvadratické prosívání, Obecné prosívání číselného pole, Lenstrova faktorizace eliptické křivky, Pollardova p-1 faktorizace, Pollardova rho faktorizace, Fermatova factorizace.

Keywords

CUDA, OpenCL, integer factorization, Quadratic sieve, General number field sieve, Lenstra's elliptic curve factorization, Pollard p-1 factorization, Pollard's rho factorization, Fermat factorization.

Citace

Jiří Podhorský: Integer Factorization on the GPU, diplomová práce, Brno, FIT VUT v Brně, 2014

Integer Factorization on the GPU

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Ivana Homoliaka

.....
Jiří Podhorský
May 26, 2014

Poděkování

Rád bych poděkoval autorům knížek a článků, kteří jsou uvedeni v bibliografii této práce, a Ing. Ivan Homoliakovi za jeho rady, jak napsat tuto práci nejlépe.

Contents

Chapter 1

Introduction

This work is about analysis of integer factorization, which means decomposition of composite integer value to prime numbers. Prime number is a number which can be divided only by itself and with number one. This work is about acceleration of integer factorization algorithms by running parts of them parallel on graphic card or on more threads on CPU.

Factorization is used mostly in RSA algorithm or other algorithms, which use multiplication of primes. Algorithms in this work can be used for breaking asynchronous RSA algorithm.

Deficiencies of factorization are in calculation of prime numbers, from which consist composed number. That's why asynchronous algorithm RSA can be used only with large composed numbers, where solving takes a lot of time. Using parallel computing can decrease this amount of time, because some parts of algorithm, which can be parallelized are computed in the same time. In this time, there can be used to compute more processors with more cores or graphic card, which uses parallel evaluation in base.

I choose this topic, because I have been fascinated with breaking a passwords and also parallel computing with graphic card, which is more and more used in this time. Till now, I only had previous experiences with threads computing on CPU and this is possible choice how to continue - using parallel computing on graphic card instead of processor unit. RSA is the most used algorithm for asynchronous encryption and can be break by quantum computers, which uses the principle of superposition and which decrease time of evaluation from exponential complexity to quadratic, because state space exploration is parallel. Then, elliptic curves algorithm will be used for encryption. Until that time, there is a challenge to compute factorization as fast as possible.

1.1 Goals of project

Main goal of the project is implementation of parallel algorithm, which will perform factorization effectively on any integer number using GPU or CPU. This algorithm will be then compared with the all serial algorithms, described in this work. There will be selected a sufficient length of key in dependency on result of time, in which algorithm found the solution.

1.2 Structure of the project

This work contains a few chapters

- chapter „Factorization methods“ is about most famous factorization algorithms. One of them will be used for acceleration with graphic card. There is explained principles of simple algorithms like Fermat factorization, Pollard's rho Algorithm or Pollard p-1 Factorization to more complicated methods like Lenstra's Elliptic Curve Factoring Method. At the last is explained the most difficult algorithms like Quadratic sieve or General number field sieve,
- chapter „Graphic acceleration“ is about frameworks for graphic cards, which accelerates program computing by using computing power of GPU and CPU. There are two most famous algorithms like OpenCL, which is supported on many platforms, or CUDA, which is supported only on NVIDIA platform,
- chapter „Summary of theory“ summarizes all information from chapter „Factorization methods“ and thinks about it. It created the first step to design of future layout of algorithm to most effective running on graphic card,
- chapter „Design of application“ is about design of application, all function, which will be included. There is summarized all information about algorithms from previous chapter with commentary, how they will work in parallel environment,
- chapter „Analyze results of application“ will contain specification the tests and all information gathered from algorithm are analyzed and summarized in complete master's work. Then will be created a result from the information. At the end of the work, there will be selected a sufficient length of RSA key dependent on algorithms computing time,
- chapter „Conclusion“ is a last chapter, which contains evaluations of achievements and work of himself.

Chapter 2

Factorization methods

In this chapter are specified most famous algorithms for solving factorization problem.

Integer factorization is a technique, which takes a composite number and returns primes, which comprise the original composite number. If number is a prime, factorization returns the number as a prime [12].

Trivial method for factorization is trying to divide the input number by predecessor of the input number and find out which predecessors divides the input number. Non-primes numbers must be divided again. When remain only primes which divides the number, the factorization process ends.

This method can be improved by finding out the primes in the array of numbers from 1 to $\sqrt{N}$, which consists only odd numbers. This method spends a lot of time on finding a primes. That's why there are a lot of famous algorithms, which are faster.

2.1 Factorization interpretation

In factorization exists two meanings:

- $x^2 + x + 3$ is a prime polynomial - having no factors except itself or one,
- 20 is prime to 21(foll by to) - having no common factors. Common factors are primes of which are consist both numbers.

2.2 Fermat Factorization

Discovered by mathematician Pierre de Fermat in the 1600s[25]. Understands composite number N as the difference of squares:

$$N = x^2 - y^2 \tag{2.1}$$

Difference can be rewritten as:

$$N = (x + y)(x - y) \tag{2.2}$$

Assuming that s and t are nontrivial odd factors of N such that $st = N$ and $s \leq t$, there can be find x and y such that $s = (x - y)$ and $t = (x + y)$. By solving this equation, can be found that $x = \frac{s+t}{2}$ and $y = \frac{t-s}{2}$, where x and y must be integers, because the difference between any two odd numbers is even and an even number is divisible by two. Since $s > 1$

and $t \geq s$, so it have to apply $x \geq 1$ and $y \geq 0$. For particular x, y satisfying $s = (x - y)$ and $t = (x + y)$, $x = \sqrt{N + y^2}$ is known and hence $x \geq \sqrt{N}$. Also, $x \leq \frac{s+t}{2} \leq \frac{2t}{2} \leq N$.

Let choose $x_1 = |\sqrt{N}|$ and $x_{i+1} = x_i + 1$. For each i, check whether $y_i = \sqrt{x_i^2 - N}$ is an integer and whether $(x_i + y_i)$, $(x_i - y_i)$ are nontrivial factors of N. If both of these conditions hold, the nontrivial factors is returned. Otherwise, continue to the next round with i and exit once when $x_i = N$.

```
function fermatFactor(N)
  for x from ceil(sqrt(N)) to N
    ySquared := x * x - N
    if isSquare(ySquared) then
      y := sqrt(ySquared)
      s := (x - y)
      t := (x + y)
      if s <> 1 and s <> N then
        return s, t
      end if
    end if
  end for
end function
```

isSquare(z) returns true if z is a square number, otherwise returns false.

2.3 Pollard's rho Algorithm

Factor a composite number N by iterating a polynomial modulo N. It was published by J.M. Pollard in 1975. First, construction of the sequence:

$$x_0 \equiv 2 \ (mod \ N) \tag{2.3}$$

$$x_{n+1} \equiv x_n^2 + 1 \ (mod \ N) \tag{2.4}$$

This sequence will eventually become periodic. The length of the cycle is less than or equal to N.

Proof by contradiction: assume that the length L of the cycle is greater than N. But there are only N distinct x_n values in the cycle of length $L > N$, so there must exist two x_n values. These two values are congruent and can be used as initial values of a cycle with length less than or equal to N. Probabilistic arguments show that the expected time for this sequence $(mod \ N)$ to fall into a cycle and expected length of the cycle are both proportional to $\sqrt{N}$, for almost all N [28]. It was found that function $f(n) = x_n^2 + 1$ works well in practice. But there can be used other initial values and iterative functions, which can have similar behavior under iteration.

Assume that s and t are nontrivial factors of N such that $st = N$ and $s \leq t$. Now suppose that non-negative integers i was found, j with $i < j$ such that $x_i \equiv x_j \ (mod \ s)$ but $x_i \not\equiv x_j \ (mod \ N)$. Since $s|(x_i - x_j)$ and $s|N$, so there is $s|gcd(x_i - x_j, N)$. By assumption $s \geq 2$, thus $gcd(x_i - x_j, N) \geq 2$. There is $gcd(x_i - x_j, N)|N$ from definition. So $N \nmid (x_i - x_j)$ and thus that $N \nmid gcd(x_i - x_j, N)$. Then $N \nmid gcd(x_i - x_j, N)$, $gcd(x_i - x_j, N) > 1$ and $gcd(x_i - x_j, N)|N$. Hence $gcd(x_i - x_j, N)$ is a nontrivial factor of N.

Goal is find i, j such that $x_i \equiv x_j \,(mod\, s)$ and $x_i \not\equiv x_j \,(mod\, N)$. The sequence $x_n \,(mod\, s)$ is periodic with the length of the cycle proportional to $\sqrt{s}$. The sequence should be compared to x_{2n} for $n = 1, 2, 3, ...$, according to Pollard x_n. Now for each n, there is need to check whether $gcd(x_n - x_{2n}, N)$ is a nontrivial factor of N. If $gcd(x_n - x_{2n}, N)$ is a trivial factor of N, the iterative process will be repeated. When factor is found, the process ends, but if no factor is found, the algorithm doesn't terminate [2].

```
function pollardRho(N)
# Initial values x(i) and x(2*i) for i = 0.
  x_0 := 2
  x_{2*i} := 2
  do
    # Find x_{i+1} and x_{2*(i+1)}
    x_{iPrime} := x_i^2 + 1
    x_{2iPrime} := (x_{2i}^2 + 1) ^ 2 + 1
    # Increment i: change running values for x(i), x(2*i).
    x_i := x_{iPrime} (mod N)
    x_{2i} := x_{2iPrime} (mod  N)
    s := gcd(x_i - x_{2i}, N)
    if s <> 1 and s <> N then
      return s, N/s
    end if
  end do
end function
```

2.3.1 Brent's Factorization Method

An improvement Pollard's rho algorithm, published by R. Brent in 1980 [26]. Instead of comparing x_n with x_{2n} for $n = 1, 2, 3, ...$, x_n to x_m will be compared, where m is the largest integral power of 2 less than n.

```
function brentFactor(N)
  # Initial values x(i) and x(m) for i = 0.
  xi := 2
  xm := 2
  for i from 1 to infinity
    # Find x(i) from x(i-1).
    xi := (xi ^ 2 + 1) (mod N)
    s := gcd(xi - xm, N)
    if s <> 1 and s <> N then
      return s, N/s
    end if
    if integralPowerOf2(i) then
      xm := xi
    end if
  end do
end function
```

integralPowerOf2(z) returns true if z is an integral power of 2, otherwise false. In terms of more efficient operations, integralPowerOf(z) is true if and only if $(z\&(z-1))$ is zero, where & is the bitwise AND operation.

Theorem. If z is a positive integer, then z is an integral power of 2 if and only if $z\&(z-1) = 0$, where $a\&b$ denotes the bitwise AND operation of a and b.

Proof. Let there be d binary bits in z, and let $(\cdot)_i$ be an operator which gives the ith binary bit of $(\cdot)$, where $i = 1$ is the least significant bit.

If z is an integral power of 2, then clearly $z_k = 0$ for $k = 1, 2, ..., d-1$ and $z_d = 1$. Because there is $z - 1 < z$, then clearly $(z-1)_d = 0$. Using the truth table for the logical AND operator, $(z\&(z-1))_k$ must be 0 for $k = 1, ..., d$. Hence $(z\&(z-1))_k = 0$.

If z is not an integral power of 2, $z_d = 1$. Let α be the largest integral power of 2 that is less than z. Then $z > \alpha$, hence $z - 1 \geq \alpha$, and thus $(z-1)_d = \alpha_d = 1$. When the logical AND operator at bit d is used, it follows $(z\&(z-1))_d = 1$, hence $(z\&(z-1))_k \neq 0$. Therefore, z is an integral power of 2 if and only if $z\&(z-1) = 0$.

2.4 Pollard p-1 Factorization

Published by J. M. Pollard in 1974 [27]. It is based on Fermat's little theorem, which states:

„If p is prime, a is a natural number, and $p \nmid a$, then $a^{p-1} \equiv 1 \ (mod\ p)$."

Suppose there is a positive integer $k \geq 1$ and a prime $p > 2$ such that $(p-1)|k!$. Now Fermat's little theorem with $a = 2$ can by apply:

$$2^{p-1} \equiv 1 \ (mod\ p) \tag{2.5}$$

But since $(p-1)|k!$, $k! = (p-1)q$ for some positive integer q can be written. Then:

$$2^{k!} \equiv (2^{p-1})^q \equiv 1^q \ (mod\ p) \equiv 1 \ (mod\ p) \tag{2.6}$$

Hence $p|2^{k!}-1$. If N is an integer which has nontrivial prime factor p, then p also divides $2^{k!} - 1 + Nt$ for all integers t. There can computed $x_k \equiv 2^{k!} - 1 \ (mod\ N)$ for $k = 1, 2, 3, ...$, and for each x_k check whether there exists an integer $r_k = gcd(x_k, N)$ which divides both x_k and N. If $(p-1)|k!$, then there is known $p|x_k$ and hence r_k is a nontrivial factor of N. If r_k is not a nontrivial factor of N, then it is a trivial factor of N, i.e. $r_k = 1$ or $r_k = N$. The algorithm is then:

Compute $r_k = gcd(2^{k!} - 1, N)$ for $k = 1, 2, 3....$ If $r_k \notin \{1, N\}$, then r_k is a nontrivial factor and all is done.

For efficiency purposes, $2^{k!} \equiv (2^{(k-1)!})^k \ (mod\ N)$ can be written, so that if $2^{(k-1)!}$ is known $(mod\ N)$, $2^{k!}$ can be computed by a single modular exponentiation operation.

```
function pollard_p1(N)
  # Initial value 2^(k!) for k = 0.
  two_k_fact := 1
  for k from 1 to infinity
    # Calculate 2^(k!) (mod N) from 2^((k-1)!).
    two_k_fact := modPow(two_k_fact, k, N)
    rk := gcd(two_k_fact - 1, N)
    if rk <> 1 and rk <> N then
```

```
      return rk, N/rk
    end if
  end for
end function
```

Here $modPow(a, b, m)$ returns the least non-negative integer y such that $a^b \equiv y \ (mod\ m)$. This function is known as modular exponentiation, which is for big integers operations, there is efficient algorithm for it.

Write b in terms of its binary digits $b_0, ..., b_{n-1}$, so $b = b_0 2^0 + b_1 2^1 + ... + b_{n-1} 2^{n-1}$ and see a^b can be rewritten as

$$a^b = a^{b_0 2^0} a^{b_1 2^1} \cdot ... \cdot a^{b_{n-1} 2^{n-1}} = (a^{2^0})^{b_0} (a^{2^1})^{b_1} \cdot ... \cdot (a^{2^{n-1}})^{b_{n-1}}. \tag{2.7}$$

Any k, $(a^{2^k})^{b_k}$ is simply 1 if $b_k = 0$ or otherwise a^{2^k}. Then:

$$a^b = \prod_{k=0, b_k \neq 0}^{n-1} a^{2^k} a^{2^{k+1}} = a^{2 \cdot 2^k} = (a^{2^k})^2 \tag{2.8}$$

There can be constructed an algorithm which returns the least non-negative integer y such that $a^b \equiv y \ (mod\ m)$, via a process of repeated squaring.

2.5 Lenstra's Elliptic Curve Factoring Method

Equation is a prime polynomial, if has no factors except itself or one. $(x^2 + x + 3)$[4]. Number is prime to number, if both of these numbers have no common factors (20 is prime to 21).

Let a and b be integers such that $4a^3 + 27b^2$ is prime to 6, and K a field of characteristic at least 5. Put

$$E(K)_{\neq 0} := \{x = (x_1, x_2) \in K^2 | x_2^2 = x_1^3 + a x_1 + b\}, \tag{2.9}$$

and let $E(K)$ be the disjoint union of $E(K)_{\neq 0}$ and the singleton $\{0\}$, where 0 is a formal symbol. Define $z \in E(K)$ for $x, y \in E(K)_{\neq 0}$ as follows.

$$\text{If } x_1 = y_1 \text{ and } x_2 = -y_2, \text{ then } z = 0. \tag{2.10}$$

Otherwise, put

$$z_1 := \lambda^2 - x_1 - y_1, \ z_2 := \lambda(x_1 - z_1) - x_2, \tag{2.11}$$

where

$$\text{if } x = y, \ \lambda = \frac{3x_1^2 + a}{2x_2} \text{ and if } x \neq y, \ \lambda = \frac{x_2 - y_2}{x_1 - y_1} \tag{2.12}$$

There is a unique structure of additive Abelian group on $E(K)$ such that 0 is the neutral element and $x + y = z$ for all $x, y \in E(K)_{\neq 0}$, z was defined above.

2.5.1 Description

Let n be a large composite positive integer prime to 6 [5]. Denote by $\equiv$ the congruence $(mod\ n)$ and choose integers $a, b, y_{0,1}, y_{0,2}$ satisfying condition:

$$gcd(4a^3 + 27b^2, n) = 1, \quad y_{0,2}^2 \equiv y_{0,1}^3 + a y_{0,1} + b, \quad 0 \leq y_{0,1}, y_{0,2} < n. \tag{2.13}$$

Let p be an (unknown) prime divisor of n, and write $x_{0,1}$ and $x_{0,2}$ for the residues $(mod\, p)$ of $y_{0,1}$ and $y_{0,2}$. In particular, $x_0 := (x_{0,1}, x_{0,2})$ is a nonzero element of $E(\mathbb{F}_p)$ [1]. Pick a positive integer k and compute kx_0 as efficiently by calculating successively:

$$x_0 + x_1,\ x_2 + x_3,\ ...,\ x_{2r} + x_{2r+1} = kx_0, \tag{2.14}$$

where $x_1 = x_0$, and each x_{2i} is equal to x_0 or to $x_{2j} + x_{2j+1}$ some $j < i$, and similarly for x_{2i+1}.

Compute $x_{2s} + x_{2s+1}$, for $1 \leq s \leq r$ [11]. The inductive assumption for stage $s-1$ is: Let x be one of the elements

$$x_0, \quad x_0 + x_1, \quad x_2 + x_3, \quad ..., \quad x_{2s-2} + x_{2s-1}. \tag{2.15}$$

There are three possible outcomes for stage s:

1. $x' := x_{2s} + x_{2s+1}$ is zero,
2. x' is nonzero and integers y'_1, y'_2, which satisfy $0 \leq y_1,\ y_2 < n$ and are representatives $(mod\, n)$ of the coordinates of x, can be computed,
3. a nontrivial divisor of n is found,

Prove 1. outcome: Assume that x_{2s} and x_{2s+1} are nonzero and that there is, for $i = 2s,\ 2s+1$ and $j = 1, 2$, integers $0 \leq y_{i,j} < n$, which are representatives $(mod\, n)$ of the coordinates of x_i. If $y_{2s,1} \equiv y_{2s+1,1}$ and $y_{2s,2} \equiv -y_{2s+1,2}$, and x' equals zero.

Prove 3. outcome: Assume $y_{2s,1} \not\equiv y_{2s+1,1}$ or $y_{2s,2} \not\equiv -y_{2s+1,2}$ and compute the gcd of $y_{2s,2}$ and n, and the gcd of $y_{2s,1} - y_{2s+1,1}$ and n. If one of these gcd is greater than one, there is the sought-for nontrivial divisor of n.

Prove 2. outcome: Otherwise, x is nonzero and representatives $(mod\, n)$ of its coordinates can be computed.

2.6 Quadratic sieve

Quadratic sieve was invented by Carl Pomerance in 1981. The QS was the fastest known factoring algorithm until the Number field sieve was discovered in 1993. Still the QS is faster than the Number field sieve for numbers up to 110 digits long.

2.6.1 Principle

If n is the number to be factored, the QS looking for two numbers x and y such that $x \not\equiv \pm y\ (mod\, n)$ and $x^2 \equiv y^2\ (mod\, n)$. This would imply that $(x-y)(x+y) \equiv 0\ (mod\, n)$ and $(x-y, n)$ will be simply computed using the Euclidean Algorithm to see if this is a nontrivial divisor. There is at least a $\frac{1}{2}$ chance that the factor will be nontrivial[7].

First step in doing so is to define function

$$Q(x) = (x + \lfloor \sqrt{n} \rfloor)^2 - n = \tilde{x}^2 - n, \tag{2.16}$$

and compute $Q(x_1)$, $Q(x_2)...Q(x_k)$. How to determine the x_i is explained in the next section. From the evaluations of $Q(x)$, there must be picked a subset such that $Q(x_{i_1})Q(x_{i_2})...Q(x_{i_r})$, which must be a square, y^2. Note that for all x, $Q(x) \equiv \tilde{x}^2\ (mod\, n)$. So then there is:

$$Q(x_{i_1})Q(x_{i_2})...Q(x_{i_r}) \equiv (x_{i_1}x_{i_2}...x_{i_r})^2\ (mod\, n). \tag{2.17}$$

This imply to $x^2 \equiv y^2\ (mod\, n)$. And when the condition $x \not\equiv \pm y\ (mod\, n)$ holds, there are factors of n.

2.6.2 Setting up a Factor Base and a Sieving Interval

The argument x_i needs to be determined by an efficient way and a product of the $Q(x_i)$ must be a square. Exponents of the prime factors, which compose the product, need to all be even, to check to see if the product is a square. Each of the $Q(x_i)$ must be factored, hence they must be small and to factor over factor base. Factor base is fixed set of small prime numbers (including -1). The number x must be selected close to 0 (to make $Q(x)$ small), so there will be set a bound M and only consider values of x over the sieving interval $[-M, M]$. Alternatively, $Q(x) = x^2 - n$ can be defined and let the sieving interval be $[\lfloor\sqrt{n}\rfloor - M, \lfloor\sqrt{n}\rfloor + M]$.

If x is in this sieving interval, and if some prime p divides $Q(x)$, then

$$(x - \lfloor\sqrt{n}\rfloor)^2 \equiv n \ (mod\, p), \tag{2.18}$$

so n is a quadratic residue $(mod\, p)$ and $n \neq 0$. So the primes in the factor base must be primes such that the Legendre symbol

$$(\frac{n}{p}) = 1 \tag{2.19}$$

and they should be less than some bound B, which depends on the size of n.

2.6.3 Sieving

There is set of primes for factor base, so numbers x from the sieving interval can be taken and $Q(x)$ can be calculated. Factor base must completely factor Q(x), then smoothness is found. If it does not, Q(x) is thrown out and the next element of sieving interval can be checked. If there is a large factor base, though, it will be better work with the entire sieving interval at once or in parallel - each processor would work over a different subinterval [9].

If p is a prime factor of $Q(x)$, then $p|Q(x + p)$. Conversely, if $x \equiv y \ (mod\, p)$, then $Q(x) \equiv Q(y) \ (mod\, p)$. So for each prime p in the factor base can be solved equation:

$$Q(x) = s^2 \equiv 0 \ (mod\, p), \ x \in \mathbb{Z}_p. \tag{2.20}$$

This can be solved using the Shanks-Tonelli Algorithm. Two solutions s_{1p} and $s_{2p} = p - s_{1p}$ will be obtained. Then those $Q(x_i)$ with the x_i in the sieving interval are divisible by p when $x_i = s_{1p}$, $s_{2p} + pk$ for some integer k. There are two ways, how to continue:

1. Depending on the size of your memory, take a subinterval and put $Q(x_i)$ in an array for each x_i in the subinterval. For each prime p, start at s_{1p} and s_{2p} and divide out the highest power of p possible for each array element in arithmetic progression, recording the appropriate powers $(mod\, 2)$ of p in a vector. There is one vector for each of the factorable $Q(x_i)$ and each entry corresponds to a unique prime in the factor base. Once all the primes have had their turn sieving the interval, those array of elements, which have now value 1 (so all elements in array have value 1, so it cannot be divided more in integer numbers), are those that factor completely over the factor base. Then the vector of powers of the primes can be put into a matrix A. This process will be repeated until there is enough entries in A to continue. When there are negative numbers in $Q(x)$, there can be add one column for sign, with 1 for negative number and 0 for positive number[17],

2. record the number of bits of the $Q(x_i)$ in an array. Subtract the number of bits of p, for every element in the particular arithmetic progressions for p. When all primes in the factor base made a step, those elements with remaining bits close to 0 are likely to be completely factorable over those primes. Is needed to take into account round-off error and the fact that many numbers are not square-free. Numbers that are not square-free, can be sieved over the subinterval a second time picking out solutions to $Q(x) \equiv 0 \ (mod\ p^2)$ and so on. When all that is done, an upper bound on the number of bits will be set. Some fully factorable numbers will slip through at this point, but it will save a lot of time. The numbers will be factored by looking at the arithmetic progressions again so it can quickly nailed down which primes divide which of the $Q(x_i)$, when meet this threshold condition. This method is less exact, but much quicker.

Most implementations of the QS do not resieve the interval looking for powers of primes. If isn't resieved with powers of primes, the threshold value becomes very important and powers of 2 becomes more significant. If $Q(x) = r^2 - n$ and r is odd, then $2|Q(x)$. There is worked with n that a higher power of 2 always divides $Q(x)$.

So, for example if 8 must always divide $Q(x)$ (when it is even), n will be considered as $(mod\ 8)$. There is $2|Q(x)$ for $n \equiv 3, 7 \ (mod\ 8)$, $4|Q(x)$ for $n \equiv 5 \ (mod\ 8)$ and $8|Q(x)$ for $n \equiv 1 \ (mod\ 8)$. To make 8 divide $Q(x)$ every time it is even, set $n := 5n$ if $n \equiv 3 \ (mod\ 8)$, set $n := 3n$ if $n \equiv 5 \ (mod\ 8)$, and $n := 7n$ if $n \equiv 7 \ (mod\ 8)$. After sieving for prime $p = 2$, sieve for the rest of the primes, subtracting the logarithms as above. Threshold will then be

$$\frac{1}{2}\ln(n) + \ln(M) - T\ln(p_{max}) \tag{2.21}$$

where T is some value around 2 and p_{max} is the largest prime in the factor base. Silverman [23] suggested that $T = 1.5$ for factoring 30-digit numbers, $T = 2$ for 45-digit numbers, and $T = 2.6$ for 66-digit numbers.

2.6.4 Building the Matrix

If $Q(x)$ does completely factor, the exponents $(mod\ 2)$ of the primes in the factor base will be put into a vector (as described in previous section). All these vectors will be put into the matrix A, that the rows represent the $Q(x_i)$ and the columns represent the exponents $(mod\ 2)$ of the primes in the factor base.

For example, if the factor base was $\{-1, 2, 3, 13, 17, 19, 29\}$ and $Q(x) = 2 \cdot 3 \cdot 17^2 \cdot 19$, then the row corresponding to this $Q(x)$ would be $(0, 1, 1, 0, 0, 1, 0)$. Product of these $Q(x_i)$ must be a perfect square, so there have to be the sum of the exponents of every prime factor in the factor base to be even, and hence congruent to 0 $(mod\ 2)$.

There may be several ways to obtain a perfect square from the $Q(x_i)$, which is good, since many of them will not give a factor of n. There is need to find solution $Q(x_1)e_1 + Q(x_2)e_2 + ... + Q(x_k)e_k$,, where the e_i are either 0 or 1, for $Q(x_1), Q(x_2),\ldots, Q(x_k)$ at the best. So if $a_i^{\rightarrow}$ is the row of A corresponding to $Q(x_i)$, then

$$a_1^{\rightarrow}e_1 + a_2^{\rightarrow}e_2 + ... + a_k^{\rightarrow}e_k \equiv 0^{\rightarrow} \ (mod\ 2). \tag{2.22}$$

So, there is need to solve

$$e^{\rightarrow}A = 0^{\rightarrow} \ (mod\ 2), \text{ where } e^{\rightarrow} = (e_1, e_2, ..., e_k) \tag{2.23}$$

so there can be found the spanning set of the solution space via Gaussian elimination, or else method hence there is need to find at least as many $Q(x_i)$ as there are primes in the factor base. Each element of the spanning set corresponds to a subset of the $Q(x_i)$. And the subset product is a perfect square. At least half of the relations from the solution space will give a proper factor. Therefore if there are $B + 10$ values of $Q(x)$ and the factor base has B elements, there is probability of finding a proper factor at least a $\frac{1023}{1024}$. Then is need to check solution vectors to see if x_i and the corresponding product of the $Q(x_i)$ yields a proper factor of n by doing a GCD calculation (described at the beginning). When a proper factor is found (after operation, there is actually two factors), test those factors for primality. If not, then check the next element in the spanning set.

Example 1:

This is description on a simple example:

$$\text{Let } Q(x) = x^2 - n \text{ and } n = 2041$$

$$\text{for } x = 46, 47, 48, 49, 50, 51 \text{ there will be:}$$

$$Q(x) = 75, 168, 263, 360, 459, 560$$

when the subset $46, 47, 49, 51$ is chosen, result of multiplication is

$$u = 46 \cdot 47 \cdot 49 \cdot 51 \equiv 311 \ (mod\ 2041),$$

there is equation

$$u^2 \equiv v^2 \ (mod\ n),$$

$Q(x)$ can be decomposed to:

$$\begin{aligned} 75 &= 3 \times 5^2 \\ 168 &= 2^3 \times 3 \times 7 \\ 360 &= 2^3 \times 3^2 \times 5 \\ 560 &= 2^4 \times 5 \times 7 \end{aligned}$$

and result of multiplication for chosen subset $Q(x)$ is

$$v = 2^5 \cdot 3^2 \cdot 5^2 \cdot 7 \equiv 1416 \ (mod\ 2041).$$

So, there is factor base $(2, 3, 5, 7)$ - that's a set of used primes.
Then from these numbers:

$$\begin{aligned} Q(75) &\equiv (0, 1, 0, 0) \ (mod\ 2) \\ Q(168) &\equiv (1, 1, 0, 1) \ (mod\ 2) \\ Q(360) &\equiv (1, 0, 1, 0) \ (mod\ 2) \\ Q(560) &\equiv (0, 0, 1, 1) \ (mod\ 2) \end{aligned}$$

A matrix can be build:

$$A = \begin{array}{c} \\ Q(75) \\ Q(168) \\ Q(360) \\ Q(560) \end{array} \begin{array}{c} \begin{array}{cccc} 2 & 3 & 5 & 7 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Every column has even count of ones and $u \not\equiv v \ (mod\ n)$, so a factor by $GCD(u-v,n)$ can be computed.

2.6.5 The Multiple Polynomial Quadratic sieve Variant (MPQS)

The MPQS uses several polynomials instead of $Q(x)$ in the algorithm and was first suggested by Peter Montgomery. By using several polynomials, the sieving interval can be made much smaller, which makes $Q(x)$ smaller, which in turn will mean that a greater proportion of values of $Q(x)$ completely factor over the factor base. These polynomials are all of the form:

$$Q(x) = ax^2 + 2bx + c. \tag{2.24}$$

Let a be a square. Then choose $0 \leq b < a$ so that $b^2 \equiv n \ (mod\ a)$. So n must be a square $(mod\ q)$ for every prime $q|a$. Then a will be chosen with a known factorization such that $(\frac{n}{q}) = 1$ for every $q|a$. Lastly, c will be chosen, so that $b^2 - 4ac = n$. When an $Q(x)$ is found that factors well, then

$$aQ(x) = (ax)^2 + abx + ac = (ax+b)^2 - n. \tag{2.25}$$

From this

$$(ax+b)^2 \equiv aQ(x) \ (mod\ n). \tag{2.26}$$

$Q(x)$ must be square, because a is square.

Let be sieving interval $[-M, M]$. There is important to optimize M and the value of $Q(x)$ over this interval.

Easy way is to determine coefficients so that the minimum and maximum values of $Q(x)$ on $[-M, M]$ have roughly the same magnitude, but be opposite in sign. Minimum is at $x = -\frac{b}{a}$. Since $0 \leq b < a$, $-1 < -\frac{b}{a} \leq 0$, and $Q(-\frac{b}{a}) = -\frac{n}{a}$ was chosen. So maximum is at $-M$ or M, and is roughly $\frac{a^2M^2-n}{a}$. Because there is need to do this with $\frac{n}{a}$, $a \approx \frac{\sqrt{2n}}{M}$ is chosen. This method is the cost of switching polynomials. Pomerance [16] says if the cost of switching polynomials is about $25-30\%$ of the total cost it would be disadvantageous to use this method. There is need new coefficients, when changing a polynomial, but for each new polynomial is also need to solve $Q(x) \approx 0 \ (mod\ p)$ for each prime p in the factor base. That is the heaviest load in switching polynomials.

Harder way is a scheme self-initialization, which can reduce the cost of switching polynomials. Principle is to fix the constant a in several of the polynomials. There is still need $a \approx \frac{\sqrt{2n}}{M}$, so let a be the product of k primes.Each prime p has a magnitude of about $(\frac{\sqrt{2n}}{M})^{\frac{1}{k}}$ and satisfy $(\frac{n}{p})$. Then for b is looked such that $b^2 \equiv n \ (mod\ a)$. There are k prime factors

of a, so there must be 2^{k-1} values for b. Then last step, can be done all at once - finding solutions to $Q(x) \equiv 0 \ (mod\ p)$ for each polynomial and for each prime p in the factor base.

Advantages of this variation is reducing the size of the factor base and sieving interval and aid in parallel processing, with each processor working with a different polynomial. If each processor generates its own polynomial(s), then it can work fairly independently, only communicating with the central server when it has sieved the whole interval.

2.6.6 The Double Large Prime MPQS Variant

The Double Large Prime MPQS Variant was employed by Lenstra, Manasse, and several others in 1993 and 1994. It considers partial factorizations of the $Q(x_i)$. In the sieving process, is hanged onto $Q(x)$ and its partial factorization if there is:

$$Q(x) = \prod p_i^{e_i} L, L > 1, L \leq p_{max}^2. \tag{2.27}$$

From definition of factor above, the factor L must be prime. There can be find these partial factorizations by increasing the threshold value after sieving by $2 \cdot \ln(p_{max})$. If there is found another $Q(x)$ whose partial factorization contains L, L can be added to the factor base and the product of the two $Q(x_i)$ will have the factor L^2. Then these two factorizations can be added to the matrix A. There may be other $Q(x_i)$ which factor over this larger factor base, so those will be added in as well. This cuts the sieving time by a sixth.[19]

2.7 General number field sieve

The fastest known factoring algorithm for factoring hard composite numbers is the general number field sieve [18]. It accepts input numbers, which are not in special form. There are various special factoring algorithms that operate faster than the GNFS, for numbers of various special forms.

The GNFS algorithm operates by finding congruent squares $(mod\ n)$. More generally, all odd composite numbers can be represented as a difference of two squares, thus providing a non-trivial factorization. This can be extended like: if $x^2 \equiv y^2 \ (mod\ n)$ non-trivially (that is, $x \not\equiv \pm y \ (mod\ n)$) then non-trivial factors of n are $gcd(x - y, n)$ and $gcd(x + y, n)$. With generating random values with this relationship the probability will be less than $\frac{1}{2}$ that the resulting values have the trivial relationship. So n can be factored by generating several such values, but not with randomly generate values, but heuristically. There isn't a particular reason that the candidates generated through our process should be any different[24].

Both the General number field sieve and the Quadratic sieve use to generate values of x and y a two step process.

1. Numbers of a particular form are sieved for smoothness (and perhaps some other properties),
2. to find subsets of the smooth numbers that can be multiplied together to form the candidate congruent squares, the matrix reduction step is used.

Half of these candidates are expected to be non-trivial, hence by generating many such candidates there is a very high probability of successful finding of a non-trivial factorization.

Though the broadest outline of these two factoring algorithms is similar, they differ significantly in their detail.

The biggest difference between both algorithms is that the Quadratic sieve operates over the integers only (over $\mathbb{Z} \times \mathbb{Z}$). The General number field sieve operates over the integers and over the ring $\mathbb{Z}[\alpha]$, so over $\mathbb{Z} \times \mathbb{Z}[\alpha]$, where α is a root of an chosen polynomial $f(X)$. Then reduce using the map

$$(\mathbb{Z} \times \mathbb{Z}[\alpha]) \xrightarrow{\phi} (\mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}) \tag{2.28}$$

where the first component, which is integer number, is simply reduction $(mod\, n)$, and the second component is a homomorphism that is called ϕ_2. This makes GNFS more complex, but with it is asymptotically better then QS.

2.7.1 Selecting the ring for polynomial

At first, there is need to choose right the parameter α and an irreducible polynomial $f(X)$ (irreducible over $\mathbb{Z}$) of degree d ($f(X)$ and d are parameters). Let α be a root of this polynomial some extension of $\mathbb{Q}$. There is an easy way of representing $\mathbb{Z}[\alpha]$, so long as $f(X)$ is monic:

$$\mathbb{Z}[\alpha] \cong \mathbb{Z}[X]/f(X) \cong \mathbb{Z} \cdot 1 \oplus \mathbb{Z} \cdot \alpha \oplus ... \oplus \mathbb{Z} \cdot \alpha^{d-1} \tag{2.29}$$

It don't have to satisfy the condition that $\mathbb{Z}[\alpha] = \mathcal{O}_{\mathbb{Q}(\alpha)}$, where $\mathcal{O}$ is ring of integers. This will be explained later.

Now a homomorphism must be created, that moves values in $\mathbb{Z}[\alpha]$ into the integers, $\phi_2 : Z[\alpha] \to \mathbb{Z}/n\mathbb{Z}$. It is sufficient to establish the value of $\phi(\alpha) = m$ and then extend $\mathbb{Z}$-linearly. Then value for m so that $f(m) \equiv 0\ (mod\, n)$ must be chosen.

There are a lot of choices for f (and thus d) and m. One option can be choose d ahead of time, generally from the integers between 3 and 10, though the optimal choice for d tends toward infinity as n tends toward infinity and then choose $m = \lfloor \sqrt[d]{n} \rfloor$.

The n can be represented as a number base m, that is $n = \sum_{i=0}^{d} c_i m^i$, where c_i is some integer from 0 to $m-1$. Then let $f(X) = \sum_{i=0}^{d} c_i X^i$. It's clear, that $f(m) = n \equiv 0\ (mod\, n)$, so the congruence requirement is satisfied. Can be assumed that $f(X)$ is irreducible. Because if not, there will be a non-trivial factorization of n: if $f(X) = h(X)g(X)$ then $n = f(m) = f(m)g(m)$.

This is generally not the optimal choice for $f(X)$, but it works.

When is given a polynomial $f(x) \in \mathbb{Z}[x]$, a root $\alpha \in \mathbb{C}$ and $m \in \mathbb{Z}/n\mathbb{Z}$ such that $f(m) \cong 0\ (mod\, n)$, there exists a unique mapping (equations 2.30 and 2.31).

$$\phi : \mathbb{Z}[\alpha] \mapsto \mathbb{Z}/n\mathbb{Z} \tag{2.30}$$

$$\begin{aligned} \phi(1) &\equiv 1\ (mod\, n) \\ \phi(\alpha) &\equiv m\ (mod\, n) \\ \phi(ab) &= \phi(a)\phi(b) \\ \phi(a+b) &= \phi(a) + \phi(b) \end{aligned} \tag{2.31}$$

for all $a, b \in \mathbb{Z}[\alpha]$ The ring homomorphism ϕ leads to the desired congruent squares. If a non-empty set S with the following properties can be found (See equation 2.32). This makes the congruence (equation 2.33).

$$\begin{aligned} y^2 &= \prod_{(a,b)\in S} (a + bm) \quad : \quad y^2 \in \mathbb{Z} \\ \beta^2 &= \prod_{(a,b)\in S} (a + b\alpha) \quad : \quad \beta^2 \in \mathbb{Z}[\alpha] \end{aligned} \tag{2.32}$$

$$\begin{aligned} \phi(\beta)^2 &= \phi(\beta)\phi(\beta) \\ &= \phi(\beta^2) \\ &= \phi\left(\prod_{(a,b)\in S} (a + b\alpha)\right) \\ &= \prod_{(a,b)\in S} \phi((a + b\alpha)) \\ &= \prod_{(a,b)\in S} (a + bm) \\ &= y^2 \end{aligned} \tag{2.33}$$

When there is an extension field $\mathbb{Q}[\alpha]$ which contains an algebraic integer that is not in $\mathbb{Z}[\alpha]$. The problem that arises is that we can end up with a product of elements from S that is not in $\mathbb{Z}[\alpha]$ and this means that we cannot use the homomorphism ϕ.

It is easier to allow the product $S(x)$ to be a product in $\mathbb{Q}[\alpha]$, because by multiplying it with $f'(x)^2$ is guaranteed that it is in $\mathbb{Z}[\alpha]$, and we can multiply the rational product with $f'(m)^2$ to obtain the wanted congruent squares.

For example, the chosen polynomial can be:

$$\begin{aligned} f_1(X) = &\ 265482057982680X^6 \\ &+ 1276509360768321888X^5 \\ &- 5006815697800138351796828X^4 \\ &- 464778544717278542717726774 50X^3 \\ &+ 652543726193598939710966737189 4785X^2 \\ &- 181857793520885943567260188624348 03054X \\ &- 277565266791543881995216199713801103343120 \end{aligned}$$

2.7.2 Sieving and linear Algebra

The ring $\mathbb{Z}[\alpha]$ is specified, so the sieving process can be specified. The sieving step operates on elements of $\mathbb{Z} \times \mathbb{Z}[\alpha]$, locating values that are smooth in their respective rings.

First will be generated a universe

$$U = \{(a,b) \in \mathbb{Z} \times \mathbb{Z} : |a| \leq u, 0 < b \leq u, gcd(a,b) = 1\} \tag{2.34}$$

with parameter $u > 0$. Value of u will establish the total number of values that will be examined in the sieve, so will be chosen later. Values of the form $(a - bm)$ and $(a - b\alpha)$ will be sieved, for all $(a,b) \in U$, for smoothness, which will be described below.

Then the relations derived from the smooth integer / algebraic integer pairs within the linear algebra step will be processed.

2.7.3 Smooth Integers (Rational factor base)

An element $(x, \gamma) \in \mathbb{Z} \times \mathbb{Z}[\alpha]$ will be defined as y-smooth, if the factorization of x and $N(\gamma)$ (the norm of γ) involves only primes less than a parameter y. The choice of y is below.

Each of the y-smooth integers can be viewed as a vector in a $\mathbb{F}_2$ vector-space, where the primes act as the basis elements. Formally, the primes equal to or less than y as $p_1, p_2, ..., p_k$ will be enumerated, that is, $k = \pi(y)$, so the $(k+1)$-st prime is larger than y. The y-smooth integer $x \in \mathbb{Z}^+$ is represented by its prime factorization, $x = \prod_{i=1}^{k} p_i^{j_i}$ and then map the value to our vector space

$$\prod_{i=1}^{k} p_i^{j_i} \to \sum_{i=1}^{k} (j_i \ (mod\ 2)) \cdot p_{j_i} \cong (j_1 \ (mod\ 2), j_2 \ (mod\ 2), ..., j_k \ (mod\ 2)) \tag{2.35}$$

A subset of our selected smooth integers will be multiplied in order to construct a square, so there will be only even exponents. In this case, this is like finding a subset of vectors that can be added to get 0 or other words, there is need to look for a linear dependency within our set of vectors. Then a square integer can be constructed by multiplying the smooth integers that correspond to the vectors present in the sum that resulted in the linear dependency, when is found a linear dependency.

When the first k primes are taken, k-dimensional vector space is got. When is chosen more than k vectors, there is guaranteed to find at least one linear dependency. There could be an infinite number of values with no linear dependency, when is set no limit to a fixed set of primes, because there are an infinite number of primes.

There is y-smooth integers, so is worked with k primes, because randomly selected integers with a particular large prime factor are much more rare than integers with a particular smaller prime factor. It's more likely to happen across instances where there are several numbers with the same small prime factor in common than numbers with the same large prime number in common, when integers are selected randomly. When at least two numbers share the same prime factor, result must be a linear dependency. So small primes are more significant than the large prime factors in our search.

$(m \ (mod\ p), p)$	$(m \ (mod\ p), p)$	$(m \ (mod\ p), p)$	$(m \ (mod\ p), p)$	$(m \ (mod\ p), p)$
(1,2)	(1,5)	(9,11)	(14,17)	(8,23)
(1,3)	(3,7)	(5,13)	(12,19)	(2,29)

Table 2.1: Rational Factor Base For $n = 45, 113$

Example of rational base. Primes p up to 29 are used (table 2.1).

2.7.4 Smooth Algebraic integers (Algebraic factor base)

When is looked for $\mathbb{Z}[\alpha]$, there is need to define a way of specifying the primes of degree 1, which divide a particular $a - b\alpha$. If prime p divides $N(a - b\alpha)$, then a prime in $\mathbb{Z}[\alpha]$ over p is expected. This prime can be represented as

$$\mathfrak{p} = (p; r_p) = (p, r_p - \alpha) \tag{2.36}$$

where

$$r_p = ab^{-1} \ (mod\ p) \tag{2.37}$$

Thus, the prime of degree 1 $\mathfrak{p}$ over p through this relation can be generated, for each a and b and prime p dividing $N(a - b\alpha)$.

The full set of possible prime divisors to $a-b\alpha$ can be enumerated, so, the primes whose norms are less than y, can be only taken, enumerated as $\mathfrak{p}_1, \mathfrak{p}_2, ..., \mathfrak{p}_l$. So result is a finite list of primes to sieve over.

Upon completing our linear algebra step, there is again a set of values such that $\prod_{(a,b)\in S}(a - b\alpha) = \beta$, where the powers for all primes in the product are even. But it don't have to be square, the obstructions are:

1. If $\mathbb{Z}[\alpha] \neq \mathcal{O}_{\mathbb{Q}(\alpha)}$, then $\beta\mathcal{O}_{\mathbb{Q}(\alpha)}$ may not be the square of any ideal,
2. if $\beta\mathcal{O}_{\mathbb{Q}(\alpha)}$ is the square of an ideal, that ideal may not be principal. For example, the ideal generated by -9 is the square of an ideal in $\mathbb{Z}$, but -9 is not a square,
3. even if $\beta\mathcal{O}_{\mathbb{Q}(\alpha)} = (\gamma\mathcal{O}_{\mathbb{Q}(\alpha)})^2$ for some $\gamma \in \mathcal{O}_{\mathbb{Q}(\alpha)}$ (i.e., $\beta\mathcal{O}_{\mathbb{Q}(\alpha)}$ is the square of a principal idea), there have to be $\beta = \gamma^2$ up to multiplication by some unit of $\mathcal{O}_{\mathbb{Q}(\alpha)}$(that is, $\beta = h\gamma^2$, where h is a unit of $\mathcal{O}_{\mathbb{Q}(\alpha)}$),
4. even if $\beta = \gamma^2$ in $\mathcal{O}_{\mathbb{Q}(\alpha)}$, there may to be $\gamma \notin \mathbb{Z}[\alpha]$.

Note: Ideal is a special subset of a ring. Ideals generalize certain subsets of the integers, such as the even numbers or the multiples of 3. Addition and subtraction of even numbers preserves evenness, and multiplying an even number by any other integer results in another even number.

For (4.), problem can be quickly solved by recalling that there can be force values from $\mathcal{O}_{\mathbb{Q}(\alpha)}$ to be in $\mathbb{Z}[\alpha]$ by simply multiplying them by $f'(\alpha)$. In this case, the square root of the algebraic component's square can be forced into $\mathbb{Z}[\alpha]$ by multiplying the algebraic component by $f'(\alpha)^2$. Then the integer component can be multiplied by $f'(m)^2$, to maintain the necessary relationship between the integer and algebraic integer components. So, if γ^2 is a square in the ring of integers of $\mathbb{Q}(\alpha)$, then $f'(\alpha)^2\gamma^2$ is a square in $\mathbb{Z}[\alpha]$.

On the obstructions (1.)-(3.), the degree of the damage can be estimated. So, can be defined a filtration based on:

- Let V_{-1} be the group generated by all elements of the form $(a-b\alpha)$ with $gcd(a, b) = 1$,
- let V_0 be the subgroup of V_{-1} such that if $\beta \in V_0$, then $\beta\mathcal{O}_{\mathbb{Q}(\alpha)}$ has only even exponents at the primes of $\mathbb{Z}[\alpha]$. This subgroup includes all of the candidates that can can be produced by the sieve step,

- let V_1 be the subgroup of V_0 such that if $\beta \in V_1$, then $\beta\mathcal{O}_{\mathbb{Q}(\alpha)}$ has only even exponents at all the primes of $\mathcal{O}_{\mathbb{Q}(\alpha)}$ ($\beta\mathcal{O}_{\mathbb{Q}(\alpha)}$ is the square of an $\mathcal{O}_{\mathbb{Q}(\alpha)}$-ideal). Said alternately, this is the subgroup of V_0 that does not suffer from obstruction (1.),
- let V_2 be the subgroup of V_1 where if $\beta \in V_2$, then $\beta\mathcal{O}_{\mathbb{Q}(\alpha)}$ is the square of a principal ideal of $\mathcal{O}_{\mathbb{Q}(\alpha)}$, say $\beta\mathcal{O}_{\mathbb{Q}(\alpha)} = (\gamma\mathcal{O}_{\mathbb{Q}(\alpha)})^2$ for some $\gamma \in \mathcal{O}_{\mathbb{Q}(\alpha)}$ Said alternately, this is the subgroup of V_1 that does not suffer from obstruction (2.),
- let V_3 be the subgroup of V_2 where if $\beta \in V_3$, then (by the above) $\beta\mathcal{O}_{\mathbb{Q}(\alpha)} = (\gamma\mathcal{O}_{\mathbb{Q}(\alpha)})^2$, and $\beta = \gamma^2$. Note $V_3 = V_0 \bigcap (\mathbb{Q}(\alpha)^*)^2$. This is the subgroup of V_2 that does not suffer from obstruction (3.).

From these definitions there is the filtration $V_0 \supset V_1 \supset V_2 \supset V_3$. By construction:

- $dim_{F_2} V_0/V_1$ measures the size of the obstruction (1.),
- $dim_{F_2} V_1/V_2$ measures the size of the obstruction (2.),
- $dim_{F_2} V_2/V_3$ measures the size of the obstruction (3.).

The naive application of the sieve step produces members of V_0, but there are needed members in V_3, which can be surely converted into solutions that certainly do not suffer from obstruction (4.). Can be computed that $dim_{F_2} V_0/V_3 \leq \log n$ by using series of algebraic arguments. So, likelihood that a randomly selected candidate that lies in V_0 also lies in V_3 and there is expectation that the proportion of these that lie in V_3 to be at least $\frac{1}{\log n}$. Inelegant solution is produce quite a lot of candidates and then run some test that will tell, which of your candidates is in the desired V_3 and is thus truly a square.

If $x \in V_0$, can be verified that $x \in V_3$ by verifying that all characters $\chi : V_0/V_3 \to \mathbb{F}_2$ are trivial at x.

It can be noted that these characters themselves form a vector space of the same dimension as V_0/V_3, which is to say dimension $\leq \log n$, so it doesn't have to tried all of the characters. This implies that if could be find a spanning set for $Hom(V_0/V_3, \mathbb{F}_2)$, then candidates could be tested by using this spanning set to verify that our candidate is certainly within V_3. There isn't any obvious way of generating such a spanning set deterministically, so it must be randomly chosen from the space $Hom(V_0/V_3, \mathbb{F}_2)$ and hope for the best.

If a vector space is dimension w and $w+s$ random vectors from the space is selected, the probability that $w+s$ vectors span the vector space is $1-2^{-s}$. By choosing a sufficiently large number of random vectors from the space, it can be made this as close to 1 as desired.

So, some way of generating random quadratic characters is needed. In the integers, the Legendre symbol $(\frac{x}{p})$ accomplishes this task so long as $p \nmid x$: so there is $(\frac{x}{p}) = -1$ implies that x is not a square and if x is not a square then $(\frac{x}{p}) = -1$ for half the primes p. Thus, it can be modified this to be a quadratic character:

$$\chi_p(x) = -\frac{1}{2}\left(\left(\frac{x}{p}\right) - 1\right). \tag{2.38}$$

So, there is an analog for this function in $\mathbb{Z}[\alpha]$ by first noting that if q is a prime integer and a and b are selected as above and $\varrho = (q, r_q)$ is degree 1, can be defined a map $\pi : \mathbb{Z}[\alpha] \to \mathbb{F}_q$ as:

$$\pi(\alpha) = ab^{-1} \; (mod\; q) \tag{2.39}$$

(extended $\mathbb{Z}$-linearly), then define $\chi_\varrho(x) : \mathbb{Z}[\alpha] \to \mathbb{F}_2$ as

$$\chi_\varrho(x) = -\frac{1}{2}\left(\left(\frac{\pi(x)}{q}\right) - 1\right) \tag{2.40}$$

It can be assured that this remains well behaved by choosing primes that can't possibly be factors of x, for example primes such as ϱ where $N(\varrho) > y$.

These quadratic characters can be considered randomly distributed as a consequence of the Chebotarev density theorem, so by selecting a suitable number of these primes, there can be (very likely) span the set of quadratic characters. Example of Algebraic Factor Base (table 2.2).

(r, p) *pair*)	(r, p) *pair*)	(r, p) *pair*)	(r, p) *pair*)	(r, p) *pair*)
(0,2)	(18,31)	(2,67)	(47,79)	(28,97)
(6,7)	(19,41)	(6,67)	(73,79)	(87,101)
(13,17)	(13,43)	(44,67)	(28,89)	(47,103)
(11,23)	(1,53)	(50,73)	(62,89)	
(26,29)	(46,61)	(23,79)	(73,89)	

Table 2.2: Algebraic Factor Base For $n = 45, 113$

2.7.5 The Quadratic Character Base

Since the quadratic character base is simply a small set of first degree prime ideals of $\mathbb{Z}[\alpha]$ that don't occur in the algebraic factor base, in practice one begins searching for roots of $f(x)$ modulo primes q with q strictly greater than the largest prime p occurring in a (r_p, p) pair in the algebraic factor base. The worked example of serves as sample illustration of how the quadratic character base seen in 2.3 is computed [3].

(r, p) *pair*)	(r, p) *pair*)	(r, p) *pair*)	(r, p) *pair*)	(r, p) *pair*)
(4,107)	(8,107)	(80,107)	(52,109)	(99,109)

Table 2.3: Quadratic Character Base For $n = 45, 113$

2.7.6 The sieve

There are two ways:

1. simply check all of our candidate values against a suitable number of these quadratic characters,

2. include these quadratic characters directly into the initial sieve step.

In order to generate a sufficient supply of smooth numbers of the correct form, they will be sieved. For the integer and algebraic integer sides a factor base will be established.

This can be abstractly thought of as a set of rules that establish which terms can be allowed within the candidates that are „passed through“ the sieve, so the list of allowed primes $p_1, p_2, ..., p_k$ is included along with the possible factor -1, which can be either present

or not, and so has exponent 1 or 0, respectively. This factor base is called B_1.

For the algebraic integer side, the factor base is made up of the full set of allowable first-order primes $\mathfrak{p}_1, \mathfrak{p}_2, ..., \mathfrak{p}_l$. If there is need to avoid the final quadratic character test, $\hat{\mathrm{l}}$ distinct quadratic characters $\chi_{\varrho 1}, ..., \chi_{\varrho \hat{\mathrm{l}}}$ can be also selected, where $\hat{\mathrm{l}}$ is chosen to be appropriately large to assure a small chance of error, as members of the factor base in the sense that each character will be applied to each component and the result of the character will be stored in the same way that the power of each prime that is present is stored. Don't expect to, in any sense, „divide" by the quadratic character elements, just store the results for every quadratic character and deal with them as if they were exponents;these will be stored within the „Quadratic Character Columns" of our final matrix. This factor base is called B_2.

Now applying the sieve step. For each value in U will be populated an integer table made up of the elements $a - bm$ and an algebraic integer table made up of elements $a - b\alpha$. It can quickly established what power of which primes divide each entry and divide them out. When it has proceeded through all the primes and prime powers in the factor basis, There are left with a set table of entries that are either units or non-units. If they are non-units, they are rejected as being insufficiently smooth. If they are units, they are accepted as y-smooth. In this process, the „divided out" terms can be stored for each entry in our table, resulting in a complete factorization for each of the resulting smooth elements. Only values for (a, b) can be accepted that result in smooth values both in the integers ($(a - bm)$ is smooth) and over the algebraic integers ($(a - b\alpha)$ is smooth).

When it is obvious, which values are being accepted, these accepted values can be optionally tested by processing the extra terms in the factor base that are not associated with prime factors (the 1 term for the integer side and the quadratic characters for the algebraic integer side), noting the resulting values as if they were exponents within the quadratic character columns.

These smooth values can be identified using $U' \subset U$, the particular $(a, b) \in U$ values that passed the sieving step. The resulting vectors for each candidate are referred to as „relations", as these are the vectors the move into the linear algebra step. In practice, this calculation can be hugely parallelized to the process scales well across clusters. This seems a complicated procedure, but it can be accomplished quite efficiently, asymptotically using $u^{2+o(1)}$ operations total, where u is as defined in the beginning of the section. Example of Pairs Found During Sieving 2.4.

(r, p) *pair*)	(r, p) *pair*)	(r, p) *pair*)	(r, p) *pair*)	(r, p) *pair*)	(r, p) *pair*)	(r, p) *pair*)
(-73,1)	(-2,1)	(-1,1)	(2,1)	(3,1)	(4,1)	(8,1)
(13,1)	(14,1)	(15,1)	(32,1)	(56,1)	(61,1)	(104,1)
(116,1)	(-5,2)	(3,2)	(25,2)	(33,2)	(-8,3)	(2,3)
(17,3)	(19,4)	(48,5)	(54,5)	(313,5)	(-43,6)	(-8,7)
(11,7)	(38,7)	(44,9)	(4,11)	(119,11)	(856,11)	(536,16)
(5,17)	(5,31)	(9,32)	(-202,43)	(24,55)		

Table 2.4: (a, b) Pairs Found During Sieving

2.7.7 Linear algebra

Now, the set of relations will be formed into a matrix and find linear dependencies between the relations [10]. When such dependencies are found, Gaussian Elimination could be used. But that would ruin our ultimate asymptotic performance, because if there are t relations each of size s, this approach would result in a runtime of $O(t^2 s)$. Hence, more advanced algorithms could be used, namely the block Wiedemann or block Lanzcos algorithms, which run in time proportional to the dimension and the weight of the matrix (the sum of the hamming weight of the vectors). Matrix creation step is called the merging process.

In practice, when is need to construct a matrix that will contain a suitably high number of dependencies, it allows to accept the inevitable loss of some of the resulting linear dependencies, which could be due to:

1. The linear dependency corresponding with a trivial relation between the two squares, which has probability heuristically $< \frac{1}{2}$,
2. the expected loss of relations that correspond to algebraic numbers V_0 but not in V_3, if the quadratic characters aren't included within the relations.

This matrix should be nearly optimal for whatever algorithm has been selected to reveal the linear dependencies. Example of pairs, which occurrences in a dependency (table).

(r,p) *pair*)	(r,p) *pair*)	(r,p) *pair*)	(r,p) *pair*)	(r,p) *pair*)
(-1,1)	(104,1)	(-8,3)	(-43,6)	(856,11)
(3,1)	(3,2)	(48,5)	(-8,7)	
(13,1)	(25,2)	(54,5)	(11,7)	

Table 2.5: Pairs Occurring In a Dependency

2.7.8 Final Calculations

If there are relations, which are not included within the factor base, the quadratic character tests rule out non-square results and issue is resolved (4.). Then our candidates are in the form

$$(f'(m)^2 \prod_{(a,b)\in S} (a - bm),\ \ f'(\alpha)^2 \prod_{(a,b)\in S} (a - b\alpha)) \tag{2.41}$$

where S is a particular subset of U'. S is associated with a linear dependency identified in the linear algebra step.

First value in this tuple is definitely a square integer and the second element is very likely a square (quadratic character test is probabilistic). The square root of the integer term can be simply calculated. This square root will be x'.

To extract square root from algebraic integer can be used the methods of Montgomery [13] or Nguyen [15]. When calculation of square root y is finished, the homomorphism developed at the start can be used and then:

$$\Phi(x', y) = (x, \Phi_2(y)) \tag{2.42}$$

Where x is just the reduction of x' $(mod\ n)$. This assures:

$$x^2 \equiv \Phi_2(y)^2\ (mod\ n) \tag{2.43}$$

Then must be check, if the resulting relationship is trivial:

$$x = \pm\Phi_2(y)\ (mod\ n) \tag{2.44}$$

There is heuristic expectation that this occurs less than half the time. So having several candidates is important even at this late stage. If this congruence is not trivial, non-trivial factors of n are found by GCD algorithm:

$$gcd(n, x - \Phi_2(y)) \text{ and } gcd(n, x + \Phi_2(y)) \tag{2.45}$$

2.7.9 Differences between GNFS and SNFS (Special number field sieve)

The SNFS works on a special type of composites, namely integers on the form: $r^e - s$, for small integers r, s and integer e and the GNFS works on all types of composites. The difference between SNFS and GNFS is in the polynomial selection part of the algorithm, where the special numbers which SNFS can be applied to, make a special class of polynomials especially attractive and the work in the square root step is also more complex for the GNFS [17].

Chapter 3

Graphic acceleration

In this chapter are described two frameworks which specifies language used for acceleration on graphic cards. Every application, which can decomposed into parallel parts, can be speed up by using acceleration computing on graphic card. Application computing can consume less time in case of using this acceleration, than in a case when it uses only CPU. There are a lot of frameworks, which use GPU for computing. Next sections introduce two most famous frameworks.

3.1 OpenCL

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms.

OpenCL provides many benefits in the field of high-performance computing, and one of the most important is portability[21]. OpenCL-coded routines, called kernels, can execute on GPUs and CPUs from such popular manufacturers as Intel, AMD, Nvidia, and IBM. New OpenCL-capable devices appear regularly, and efforts are underway to port OpenCL to embedded devices, digital signal processors, and field-programmable gate arrays.

The OpenCL application itself can be written in either C or C++, the source for the application kernels is written in a variant of the ISO C99 C-language specification. These kernels are compiled via the built-in runtime compiler, or if desired, are saved to be loaded later[22].

OpenCL Architecture consists

- parallel computing for heterogeneous devices,
  - CPUs, GPUs, other processors (Cell, DSPs, etc),
  - portable accelerated code,
- defined in four parts:
  - platform Model,
  - execution Model,
  - memory Model,
  - programming Model.

3.1.1 Platform Model

Platform model is

- the model consists of a host connected to one or more OpenCL devices,
- a device is divided into one or more compute units,
- compute units are divided into one or more processing elements.

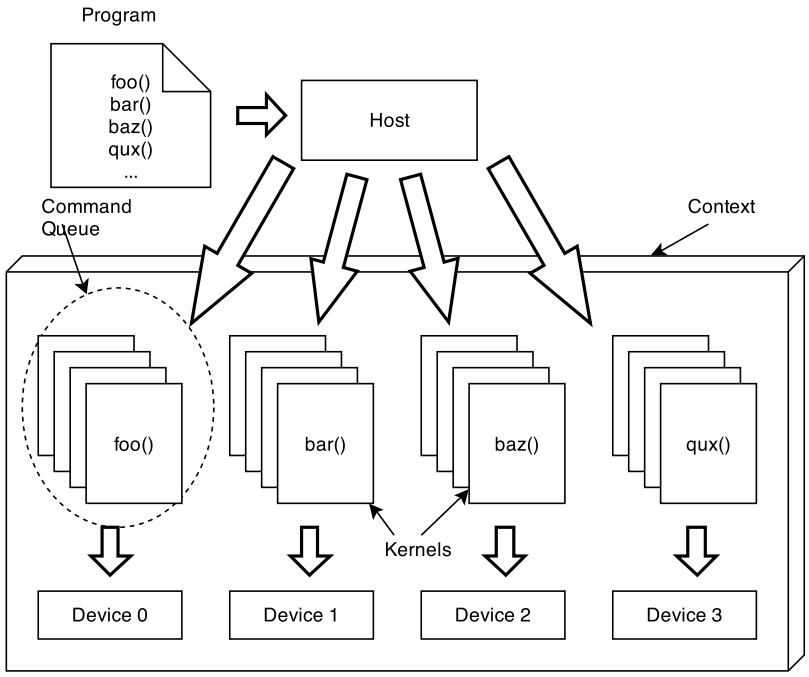


Figure 3.1: Distributing kernels to OpenCL-compliant devices.

3.1.2 Execution Model

It's host programs execute on the host and also kernels execute on one or more OpenCL devices (figure 3.1).

- Each instance of a kernel is called a work-item,
- work-items are organized as work-groups,
- when a kernel is submitted, an index space of work-groups and work-items is defined,
- work-items can identify themselves based on their work-group ID and their local ID within the work-group,
- a context refers to the environment in which kernels execute,
    - devices,

    - kernels (OpenCL functions that run on OpenCL devices),
    - program objects (The program source that implements the kernel),
    - memory objects (Data that can be operated on by the device),
    - command queues are used to coordinate execution of the kernels on the devices,
- memory commands (data transfers),
- kernel synchronization commands,
- synchronization,
- execution between host and device(s) is asynchronous,
- commands can execute in-order or out-of-order.
- Kernel
    - kernels are entry points to the device program,
    - function that is executed on the device,
    - only functions that can be called from the host,
    - every code that runs on the device, what includes kernels and non-kernel functions called by kernels, are compiled at run-time,
    - must placed on global memory or constant memory or local memory or private memory,
    - executed by one or more work-items (see figure 3.1),
- OpenCL Program
    - formed by a set of kernels, functions and declarations,
- Platform
    - the host and collection of devices managed by the OpenCL framework,
    - allow an application to share resources and execute kernels on devices in the platform,
- OpenCL Device
    - receive kernels from the host,
    - in-order and out-of-order execution are possible,
    - internally can potentially support large numbers of concurrent threads of execution,
- OpenCL Context
    - defines the entire OpenCL environment, including OpenCL kernels, devices, memory management, command-queues, etc, within which work items executes, which includes devices and their memories and command queues,
    - allows devices to receive kernels and transfer data,
- Command-Queue

- object, where OpenCL commands are enqueued to be executed by the device,
- created on a specific device in a context,
- multiple command-queues allows applications to queue multiple independent commands without requiring synchronization,
- through it each device receives kernels,

3.1.3 Programming Model

Programming model consists of:

- Data parallel
  - one-to-one mapping between work-items and elements in a memory object,
  - work-groups
    - can be defined explicitly (like CUDA) or implicitly (specify the number of work-items and OpenCL creates the work-groups),
    - exist to allow communication and cooperation between work-items,
    - reflect how work-items are organized (it's a N-dimensional grid of work-groups, with N = 1, 2 or 3),
    - equivalent to CUDA thread blocks,
    - as work-items, work-groups also have an unique ID that can be referred from the kernel,
  - work-items
    - is basic unit of work on an OpenCL device,
    - collected into work-groups and each work-group executes on a compute unit,
    - can then be queued on one to many devices to achieve very high performance and scalability,
- Task parallel
  - kernel is executed independent of an index space,
  - other ways to express parallelism: enqueueing multiple tasks, using device-specific vector types, etc,
- Synchronization
  - possible between items in a work-group,
  - possible between commands in a context command queue.

3.1.4 Memory model

Memory model consists of: (figure 3.2)

- Global memory:
  - stores data for the entire device (Accessible by all work-items),
  - generally is the largest capacity memory subsystem on the compute device,

- should also be considered the slowest memory subsystem that also has some restrictions on use, which complicates code design,
- should be considered as streaming memory - best performance will be achieved when streaming contiguous memory addresses or memory access patterns that can exploit the full bandwidth of the memory subsystem,

- Private memory:
  - stores data for an individual work-item,
  - memory is used within a work item that is similar to registers in a GPU multiprocessor or CPU core,
  - private per work-item memory,
  - fast and can be used without need for synchronization primitives,
  - it is allocated and partitioned at compile time by the JIT compiler for the given kernel and card,
  - difficult to decide how much private memory use,
  - when GPU device doesn't have cache memory, memory can be spilled to slow global memory and will cause significant performance drops,
- Local memory:
  - stores data for the work-items in a work-group,
  - OpenCL local memory is much faster than global memory – generally on-chip [14],
  - local memory is used to enable coalesced accesses, to share data between work items in a work group, and to reduce accesses to lower bandwidth global memory,
- Constant memory:
  - similar to global memory, but is read-only,
  - a read-only region of memory,
  - NVIDIA GPU devices, this is a specialized region of memory that is good for broadcast operations,
  - AMD devices, this is a region of global memory that exploits hardware optimizations to broadcast data,
- Bank conflict
  - can dramatically slow application performance,
  - for performance reasons memory subsystems are arranged in banks to increase streaming bandwidth by a factor related to the number of banks,
  - occurs when multiple threads try to simultaneously access the same memory bank at the same time,
  - bank conflicts are very device dependent.

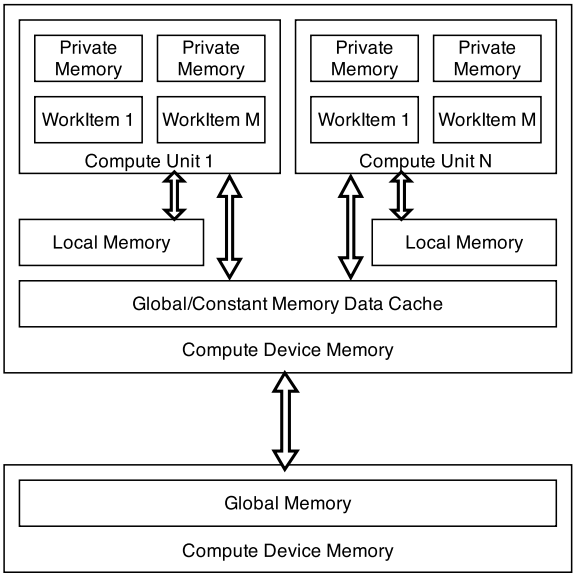


Figure 3.2: The OpenCL device model.

3.1.5 Rules to achieve high-performance

High-performance applications follow three general rules[6]. Appropriate use of the device memory space and hierarchy are critical

- get and keep the data on the GPU to eliminate PCI bus data transfer bottlenecks,
- give the GPU enough work to do,
- starting a kernel does require a small amount of overhead. However, modern GPUs are so fast that they can perform a significant amount of work while the kernel is being started,
- optimize the calculation to minimize the bottleneck in accessing the GPU memory.

3.2 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and scalable parallel programming model created by NVIDIA and a software environment for parallel computing, minimal extensions to familiar C/C++ environmental and Heterogeneous serial-parallel programming model[8].

- Device = GPU,
- Host = CPU,
- Kernel = function that runs on the device.

3.2.1 CUDA Kernels and Threads

CUDA Kernels [20] and Threads are

- parallel portions of an application are executed on the device as kernels,
    - one kernel is executed at a time,
    - many threads execute each kernel,
- Differences between CUDA and CPU threads
    - CUDA threads are extremely lightweight,
        - very little creation overhead,
        - instant switching,
    - CUDA uses 1000s of threads to achieve efficiency,
        - multi-core CPUs can use only a few.

3.2.2 Arrays of Parallel Threads

Arrays of Parallel Threads are

- a CUDA kernel is executed by an array of threads,
- all threads run the same code,
- each thread has an ID that it uses to compute memory addresses and make control decisions.

3.2.3 Thread Cooperation

Thread Cooperation is

- share results to avoid redundant computation,
- share memory accesses,
    - drastic bandwidth reduction,
- powerful feature of CUDA,
- cooperation between a monolithic array of threads is not scalable,
    - cooperation within smaller batches of threads is scalable.

3.2.4 Thread Batching

Thread Batching is

- kernel launches a grid of thread blocks,
    - threads within a block cooperate via shared memory,
    - threads within a block can be synchronized,
    - threads in different blocks cannot cooperate,
    - each block has a unique ID within a grid (block ID) and a unique ID within a block (thread ID),
- allows programs to transparently scale to different GPUs.

3.2.5 Key Parallel Abstractions in CUDA

Key Parallel Abstractions in CUDA is

- trillions of lightweight threads,
    - simple decomposition model,
- hierarchy of concurrent threads,
    - simple execution model,
- lightweight synchronization of primitives,
    - simple synchronization model,
- shared memory model for thread cooperation,
    - simple communication model,

Chapter 4

Summary of theory

This chapter summarize previous chapters. All information in this chapter are decomposed into subproblems and there are considerations about every subproblem, whether this subproblem can be decomposed into parallel parts. Then will be chosen the best algorithm, which will be parallelized.

These algorithms were presented in the most of publications I read, that's the reason, why they are such famous. They also explain problem of factorization on more levels. So, this factorization work can be read and studied from the beginning with simpler algorithms to more complicated at the end.

4.1 Fermat Factorization

This algorithm rewrites composite number as difference of squares. Is used as basic algorithm for others better algorithms. Number have to be tested by pre-evaluated primes, because this algorithm is only for odd number, not even. In this algorithm is known exactly in the beginning, how much steps must be done (It's $\sqrt{n}$). Each step is independent, so each step can be computed parallel and the result will be prime or not. Time of all parallel evaluation will be then time of the longest one (probably the biggest number). Class of complexity will be then $\mathbb{O}(k+n)$. That's why the algorithm can be easily divided into parallel blocks. But it will be slow for bigger numbers, because for each number which precedes n must exists parallel block. This count of parallel block will be huge. So, if there isn't enough blocks, each block must composed from more steps of parallel computing than one. But this will slow the computation (figure 4.1).

4.2 Pollard's rho Algorithm

This algorithm uses polynomial equation for factoring composite number. It can be better than previous one. Brent algorithm is only improvement which speed it up. At the beginning of algorithm, there is no fixed evaluation of computations or steps. Results of each compute will be used in another step. So, the algorithm can be divided into parts, but not in constant parts, which are necessary on GPU. Hence the parts of algorithm cannot be used in parallel computing.

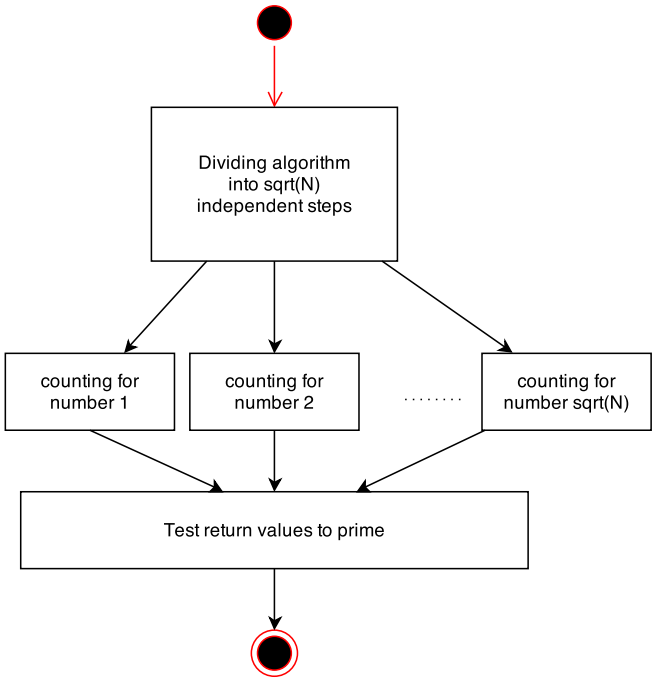


Figure 4.1: Diagram of fermat factorization in parallel form.

4.3 Pollard p-1 Factorization

This algorithm uses modulo operations like the previous ones. But unlike previous it doesn't need polynomial equation, it uses powers of base number, which can be rewrite as factorials. In terms of parallelization, this algorithm is the same as previous.

4.4 Lenstra's Elliptic Curve Factoring Method

This algorithm uses random number generator for generate 4 numbers (a, b, $y_{0,1}$, $y_{0,2}$). These numbers are used in conditions (2.13), which must be satisfied and addition points to elliptic curve, until there remains a prime. There are one part of algorithm, which can be parallelized. It depends on how many times will be generated 4 numbers, from which consist elliptic curve, and tested for nontrivial factors (figure 4.2).

That is not enough for parallel computing, because usually are chosen only few tests (x variable in this picture).

4.5 Quadratic sieve

This algorithm is based on same theory like Fermat Factorization. It is quick and can be easily parallelized. Number have to be tested by pre-evaluated primes, because this algorithm is only for odd number, not even.
First step is creating a factor base, that's fixed set of small prime numbers including -1. We can have this pre-evaluated before the algorithm starts and saved in the file.

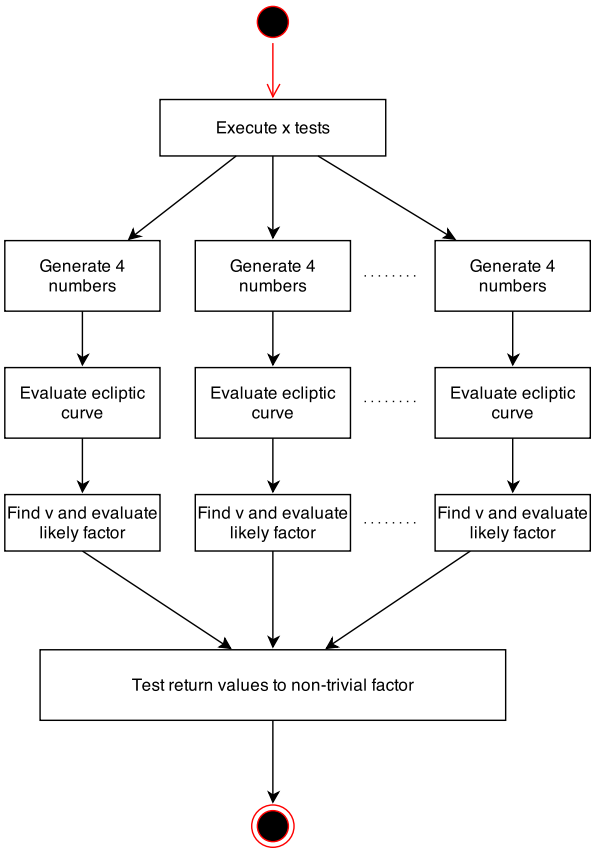


Figure 4.2: Diagram of Elliptic curve algorithm in parallel form.

Now comes the sieving step, the decomposition of $Q(x)$, is evaluated (by equation 2.16). $Q(x)$ must be factorable. One processing thread can decompose one $Q(x_i)$ or subset of $Q(x)$. We don't need to keep exponents of all primes, which composite each number. We keep only exponent $(mod\ 2)$. As a result we have a set of vectors of bool values for each taken number. If number cannot be decomposed (divided) completely, this number (and vector) will be thrown out.
Next step is building matrix and looking for linear dependencies. Now we have set of boolean vectors, which can be composed into matrix. We choose few numbers and give them into matrix. This step can be also parallelized. If every column on chosen matrix has even count of ones, we have numbers, which composites a square. This is simple looking for linear dependencies of vectors in matrix, which can be evaluated for example by *gauss elimination* algorithm.
At the end, we evaluate factors by GCD algorithm.

It is suitable algorithm for parallelization. There is two section, which can be parallelized (diagram 4.3).

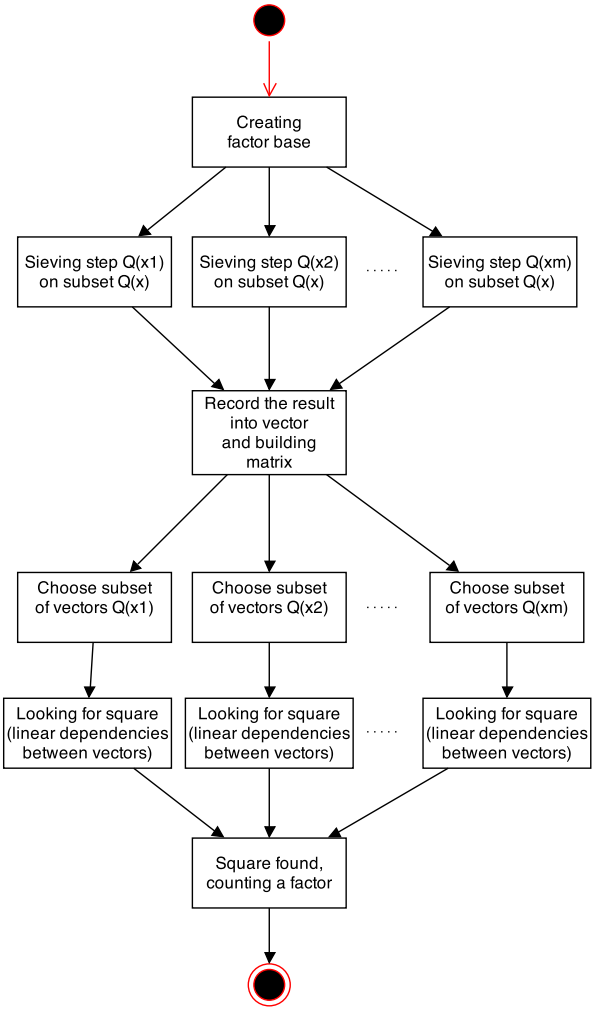


Figure 4.3: Object diagram of Quadratic sieve algorithm.

4.6 General number field sieve

This algorithm is improvement of QS, some parts seems to be almost the same, but some parts are completely different. Quadratic sieve operates over the integers only (over $\mathbb{Z} \times \mathbb{Z}$) and the General number field sieve operates over the integers and over the ring $\mathbb{Z}[\alpha]$ (so over $\mathbb{Z} \times \mathbb{Z}[\alpha]$, where α is a root of an chosen polynomial $f(X)$). So this algorithm should be better for parallel computing than QS. First step is choose polynom and its parameters. Then there are three factor basis, which must be created: Rational, Algebraic and Quadratic. Then Rational, Algebraic basis must be sieved and the result will be list of primes, which is smooth in its factor base. There are some obstruction, which can be dangerous, so with using Quadratic base some pairs, which not satisfy the conditions, will be thrown out. Then matrix will be created from results and linear dependency between rows in matrix will be found. Then will be found a factor by square root of them. This can be seen in diagram of function GNFS (figure 4.4).

This algorithm is very difficult to understand, but is more quicker than QS and provides more steps than QS, which can be parallelized.

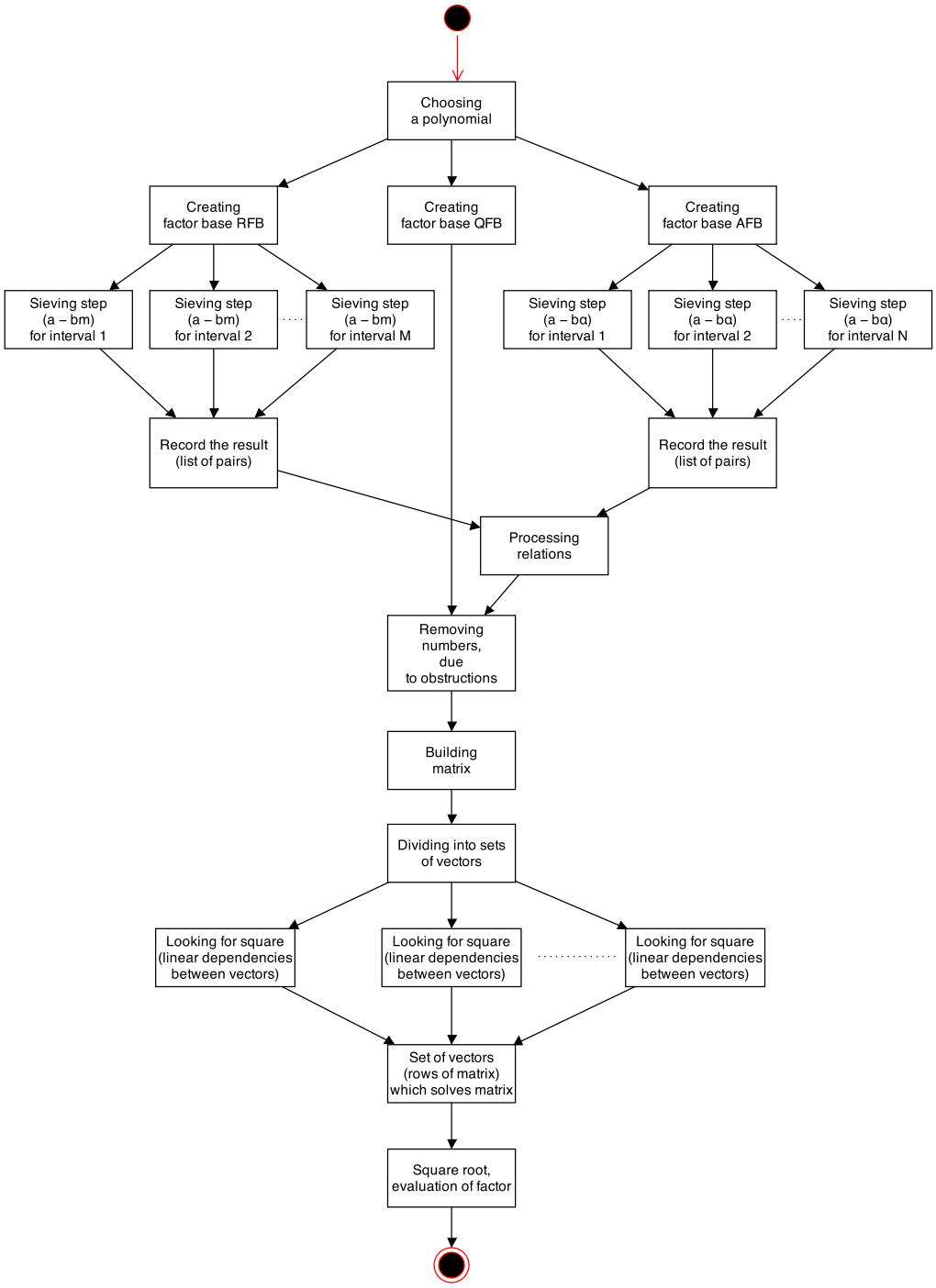


Figure 4.4: Object diagram of General number field sieve algorithm.

4.7 OpenCL and CUDA

OpenCL is supported at many devices and it's not a new language. It doesn't have threads, but working items in workgroups. We suppose the algorithm will be designed at such level, that it would be capable of decreasing a computational time of finding primes.

The main disadvantage of CUDA is that the framework works only on NVIDIA graphic card. It uses CUDA threads, which can better use the whole GPU. The calculation doesn't have to be fixed in every block. Kernels are composed of the threads. It could be faster.

This application should be work on more types of HW not only NVIDIA card, because future user don't have to own this type of graphic card. So, maybe OpenCL will be better choice.

4.8 Time complexity of algorithms

In the table 4.1 are summarized time complexities of serial algorithms discussed in previous sections. Equations in the table doesn't have any informative value in terms of parallelization. Algorithm, which is slow in serial, can be quicker in parallel. It is only just out of curiosity.

Name of algorithm	**Time complexity**	**Comment**
Fermat Factorization	$O(n^{\frac{1}{3}})$	None
Pollard's rho Algorithm	$O(n^{\frac{1}{4}})$	None
Pollard p-1 Factorization	$O(B \cdot \log B \cdot \log^2 n)$	Larger values of B (smoothness bound) make it run slower, but are more likely to produce a factor
Lenstra's Elliptic Curve Factoring Method	$O(\exp(\sqrt{2+o(1)}\sqrt{\ln p \ln \ln p}))$	p is the smallest factor
Quadratic sieve	$O(\exp(\sqrt{\log n \cdot \log \log n}))$	None
General number field sieve	$O(\exp(\frac{64}{9} \cdot b)^{\frac{1}{3}}(\log b)^{\frac{2}{3}})$	for b-bit number n

Table 4.1: Time complexity of algorithms.

4.9 Results

Quadratic sieve and General number field sieve algorithms are the most appropriate to parallel processing. Algorithms QS has many modifications, from which were selected and showed two. GNFS has also many modifications, but these are intended for special purpose. Each of these special modifications are selected for special set of numbers, which consists of primes. These modifications are much quicker, but there weren't assumed special set of numbers. So, It wasn't written here. QS is suitable algorithm, but GNFS expands it principally more, because for larger input numbers the GNFS works quicker. There is need to find out how large number is needed for safe algorithm RSA as the last part of this work, so GNFS will be a better choice.

There was decided to use OpenCL, because there was made only little related work on the field of integer factorization acceleration. Moreover, there are a lot of applications, which use CUDA. There was opportunity to find another way. If everything will work, comparing parallel application with other existing solutions can provide us some results. Another implementations like CUDA can be compared too. Then find out the best algorithm for OpenCL.

Chapter 5

Design of application

This chapter is about design of application and all function, which will be included. There is summarized all information about algorithms from previous chapter and there is discussed our considerations and suggestions on these algorithms. Decompose them into blocks of computing. There is chosen an algorithm for parallel computing acceleration and next is specified its blocks structure for implementation.

5.1 Parts of whole system

System consists of

- script in bash or other scripting language, which will build applications from source code and provided interface for communication with applications,
- implementation of parallelized application with use:
  - *General number field sieve* algorithm,
  - acceleration by OpenCL framework,
  - threads for multicore processor,
  - system has automatic settings, for use with inexperienced people,
- implementation of others unparalleled applications (other algorithms), which are described in this work (chapter 2):
  - Fermat Factorization algorithm,
  - Pollard's rho algorithm,
  - Pollard p-1 Factorization,
  - Lenstra's Elliptic Curve Factoring Method,
  - Quadratic sieve,
  - General number field sieve.

This will be classes of chosen GNFS algorithm (figure 5.1).

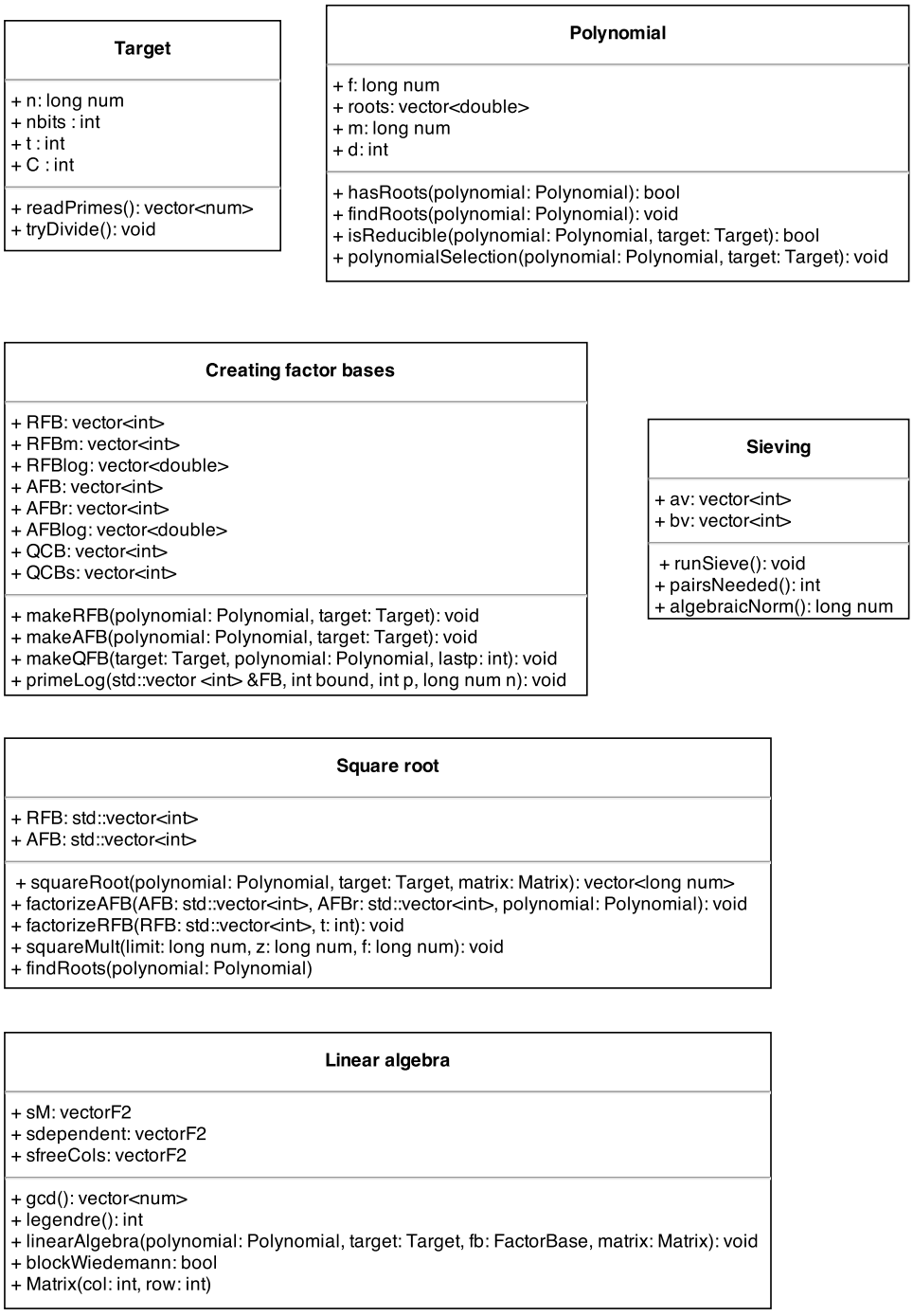


Figure 5.1: Object diagram of chosen GNFS algorithm.

5.2 Details of parallelized algorithm

Chosen algorithm are decomposed into simple parts to find, which ones are useful for parallel computing on GPU (in kernel) and which should be transformed and which should remain in unparalleled structure.

The decomposition of algorithm to components diagram is illustrated in the figure 5.2.

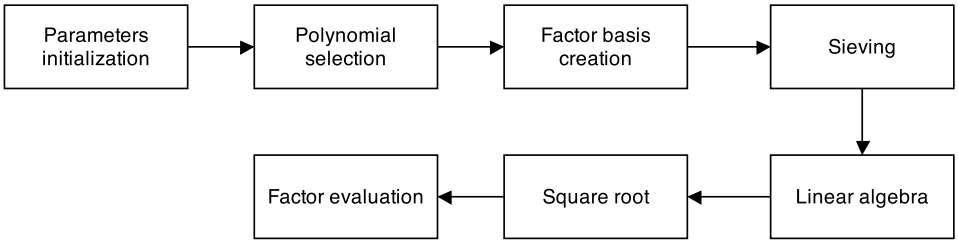


Figure 5.2: Object diagram of chosen algorithm (GNFS).

Next, each component of the diagram will be described.

5.2.1 Parameters initialization

This is first step. In this step is used a file of primes and the composed number is divided by this set of numbers. It is necessary, because of this algorithm works only with odd numbers, so even numbers have to be divided by powers of two.
This can be also parallelized by using more evaluation blocks, which will divide number with different numbers power of two. The biggest number, which can divide composed number completely without remainder, will be used.

5.2.2 Polynomial selection

The parameters of polynomial and polynomial are selected in this step. There is need to find an irreducible polynomial $f(x)$ with root m, ie. $f(m) \equiv 0 \ (mod\ n)$, $f(x) \in \mathbb{Z}[x]$. The polynomial is in form

$$f(x) = \sum_{i=0}^{d} c_i x^i$$

where c_i is some integer from 0 to $m - 1$.
This step can be parallelized, because there is looked for polynomial which satisfy the conditions. So, we can search for more polynomials here. When polynomial does not satisfy the conditions, it is discarded.

5.2.3 Factor basis creation

In this step we need to choose size for factor basis and set them up.

There is needed three factor basis:

- rational factor base,

- algebraic factor base,
- quadratic factor base.

This process can be parallelized, if there will be sufficiently large all three basis. Factor basis are sets of primes, which are used for decompose $Q(x)$ in QS algorithm and a, b in GNFS algorithm. So in this step is file of primes needed also.

5.2.4 Sieving

In this step we need to find pairs of integers (a, b) with properties:

- $gcd(a, b) = 1$,
- $a + bm$ is smooth over the rational factor base,
- $b^{deg(f)} f(\frac{a}{b})$ is smooth over the algebraic factor base.

A pair (a, b) with these properties is called a relation. The purpose of the sieving stage is to collect as many relations as possible (at least one larger than the elements in all of the basis combined). The sieving step results in a set S of relations.
This step can open the huge potential of GPU. This step have to be processed on GPU.

5.2.5 Linear algebra

In this step the results from the sieving are filtered by removing duplicates and the relations containing a prime ideal not present in any of the other relations. The relations are put into relation-sets and a very large sparse matrix over **GF(2)** is constructed. The matrix is reduced resulting in some dependencies, ie. elements which lead to a square modulo n.

There is few algorithms, which can be used for solving linear algebra:

- Gauss elimination:
  - maybe the most famous algorithm,
  - too slow in terms of time complexity.
- Block Lanczos algorithm:
  - it is very quick,
  - it cannot be used in parallel, because computational cannot be done independently.
- Block Wiedemann algorithm:
  - it can be used in parallel,
  - it is very quick.

When is used Wiedemann algorithm, there can be a little improvement in parallel algorithm.

5.2.6 Square root

In this step is used the square root.

Calculate the rational square root, ie. y with

$$y^2 = \prod_{(a,b)\in S} (a - bm).$$

Calculate the algebraic square root, ie. x with

$$x^2 = \prod_{(a,b)\in S} (a - b\alpha)$$

where α is a root of $f(x)$.

There are two recommended methods

- Montgomery method,
- Nguyen method.

These two methods seems to be very similar. Better can be Nguyen method, which is newer and can have some improvements.
It is simple to compute, so it will be better keep to it on CPU.

5.2.7 Factor evaluation

In this step, the evaluation of factors from numbers with GCD algorithm is processed. The prime p can then be found by $gcd(n, x - y)$ and $gcd(n, x + y)$ according to the equation (2.45).
This step is easy to compute, so it will be executed it on CPU.

5.3 Other implementations of algorithms

The listing of known integer factorization algorithm is depicted in the table (table 5.1). All of them use arithmetic library for large integers. Because standard format of numbers is too small, there is need to use char array. Library for acceleration are CUDA, which was described before (section 3.2) and Open MPI. Open MPI uses threads on CPU, not on GPU and threads have to be declared before (like in OpenCL), not on runtime.

Name of implementation	**used algorithm**	**Acceleration**	**arithmetic library**
msieve [1]	SIQS and GNFS	CUDA	GMP
kmGNFS [2]	GNFS	Open MPI	NTL
pGNFS [3]	GNFS	None	NTL
GGNFS [4]	GNFS	None	GMP

Table 5.1: Few others implementations of algorithm.

5.4 Results of theory

This algorithm will be theoretically the fastest on GPU, but it should be better in practice of running application. But this will be showed after some tests of application. We are now in theoretical part of work. Maybe some steps can slow down the speed of application, but this may be barely detectable, until the implementation of algorithm will be created and some tests will be performed.

5.5 Implementation of application

In this section are described implementations of all factoring algorithms in this work.

Every implementation of factoring algorithm has two modes:

- RSA - looking for first result of factoring, don't presume there is more primes than two. In QS and GNFS it also mean, the file of pre-evaluated set of primes won't be used. This mod is build especially for testing and breaking keys of RSA encrypting algorithm.
- Normal - tries to decompose composite number to primes. This method can use pre-evaluated set of primes to extract primes from composite number. But pre-evaluated numbers uses only QS and GNFS algorithms.

There are used pre-evaluated set of primes for creating factor base in factoring algorithms QS and GNFS.
For implementation was used NTL library in every algorithm and openCL library for GNFS parallel. Every algorithm was implemented in C++ and script was implemented in bash.

5.5.1 Fermat Factorization

Fermat Factorization algorithm was implemented exactly according the pseudo-code in second chapter 2.2. It contains improvement, just like all others simple algorithms. This improvement is dividing even numbers with 2 to get odd.

5.5.2 Pollard p-1 Factorization

This algorithm was implemented also according the pseudo-code in second chapter 2.4. After some experiments on testing number set, it has been observed, that better results were achieved by the base number 3 instead of 2. It was applied the powerMod function on this number after testing. It contains improvement, just like all others simple algorithms. This improvement is dividing even numbers with 2 to get odd.

5.5.3 Pollard's rho Algorithm

There was used improved implementation of Pollard's rho algorithm. Pseudo codes of Brent's Factorization Method and Pollard's rho Algorithm from second chapter 2.3 were

[1]URL `sourceforge.net/projects/msieve/`
[2]URL `kmgnfs.cti.gr/kmGNFS/Home.html`
[3]URL `pgnfs.org/`
[4]URL `www.math.ttu.edu/~cmonico/software/ggnfs/;`

tests, but because for some numbers the algorithm was stuck in infinite loop, these algorithms are valid only for explanation and understanding the algorithms. Because this algorithm like Pollard's rho Algorithm evaluate number until it found two factor numbers, the following implementation of Rabin Miller algorithm for detection primes was used in this algorithm for blocking cycling. It contains improvement, just like all others simple algorithms. This improvement is dividing even numbers with 2 to get odd.
Rabin Miller algorithm

1. if $p < 2$ return false
2. if $p! = 2$ and p is even, return false;
3. evaluate $s = p - 1$ and make from s even number by dividing power of 2.
4. Generate random number a in $< 1, p)$
5. evaluate $mod = Power(a, temp)\ (mod\ p)$
6. Try evaluate $(mod * mod)\ (mod\ p)$, and multiple s by 2 until mod is 1 or mod is $p - 1$ or s is $p - 1$
7. if all found numbers aren't even and not $p - 1$, wasn't performed enough count of tests, continue with next test point 4

5.5.4 Lenstra's Elliptic Curve Factoring Method

Implemented Elliptic curve algorithm is based on equation $y^2 = (x^3 + ax + b)(mod\ n)$. Following pseudo-algorithm was created from source code itself. So it belongs to this place. There was used two structures: elements - keeps a, b and N value, and points - keep point of curve or identity.

1. Choose limit to adding points to elliptic curve and number of tests to pass until algorithm returns factor or prime,
2. get random non-zero coordinates for the curve: point $P = (u, v)$, where $u, v\ (mod\ N)$ and then pick a random non-zero $A\ (mod\ N)$ and evaluate $B = (u^2 - v^3 - Ax)\ (mod\ N)$,
3. curve $y^2 = x^3 + ax + b$ is defined over the field K, whose characteristic we assume to be neither 2 nor 3, and points $P = (xP, yP)$ and $Q = (xQ, yQ)$ on the curve, assume first that $xP! = xQ$. Let the slope of the line $s = (yP - yQ)/(xP - xQ)$; since K is a field, s is well-defined. Then can be defined $R = P + Q = (xR, -yR)$ by:

$$\begin{aligned} s &= (xP - xQ)/(yP - yQ)\ (mod\ N) \\ xR &= s^2 - xP - xQ\ (mod\ N) \\ yR &= yP + s(xR - xP)\ (mod\ N) \end{aligned}$$

4. If $xP = xQ$, then if $yP = -yQ$, including the case where $yP = yQ = 0$, then the sum is defined as 0 - $[Identity]$. Thus, the inverse of each point on the curve is found by reflecting it across the x-axis. If $yP = yQ! = 0$, then $R = P + P = 2P = (xR, -yR)$ is given by:

$$\begin{aligned} s &= (3xP^2 + a/(2yP))\ (mod\ N) \\ xR &= (s^2 - 2xP)\ (mod\ N) \\ yR &= (yP + s(xR - xP))\ (mod\ N) \end{aligned}$$

5. adding next points, until factor will be found or the limit is reached:
   (a) Q=0 #declare identity element
   (b) until limit is reached
      i. if (m is odd) Q+=P #add next point
      ii. P+=P #change point to another
   (c) return factor
6. if there is no factor found, continue by second point.

5.5.5 Quadratic sieve

Evaluation of size of factor base is based on bits size of composite number. The normal mode uses also „Rabin Miller algorithm“ introduced in previous „Pollard's rho Algorithm“ section for the primes detection. Following pseudo-algorithm was created from source code itself. So it belongs to this place.

1. This factor base are created from file of primes. From this file are chosen only primes, which remains after dividing the composite number are quadratic residue, so it can be square root in integer numbers.
2. There is prepared set of quadratic numbers Qx in Node structure consists of:

$$Nodet; t.m = offset + i; t.q_x = sqrt(N) + t.m; t.q_x = t.q_x * t.q_x - N;$$

3. There the smooth numbers are taken from factor base. Simply, there is chosen set of numbers, in which all numbers are composed only by numbers in this set. The smooth numbers are evaluated in loop by factorization in two directions (In positive and negative), until there is enough numbers for next phase. This number is based on size of factor base, but there is useful add more. There is added 32, because this number seems to be enough. Factorization is done by Eratosthenes sieve and decompose this small numbers to prime by trial division by numbers in factor base.
   - trial division is algorithm in which the number are divided by other numbers (if division could be done in integer numbers) with numbers antecedent the number, mostly with numbers in set (2, sqrt(number)), in this algorithm the number is divided by primes from factor base.
   - Eratosthenes sieve is simple algorithm in which we take first number and all next numbers, which can be divided by this number are not a primes.

This step is better to parallel. But this algorithm had to be serial.

4. Results are given to matrix and then Gaussian elimination method is passed above the matrix. This method looked for linear dependencies. There is count of single prime in single cell, which was used in single qx number. There is need to keep only if the count is even or odd.

5. The last step is square root. There is looked for rows, with sum of columns are equal to zero ($mod\ 2$). When is found, the result numbers are added to equation and then square root give the result factor from equation: $u \not\equiv v\ (mod\ n)$, so a factor by $GCD(u - v, n)$ can be computed.

The normal version uses also „Rabin Miller algorithm“ introduced in previous „Pollard's rho Algorithm“ section.

5.5.6 General number field sieve

This algorithm is implemented like was explained in design chapter 5. Only one new class was created for composite numbers and it's properties.
The Basic Gaussian elimination method was used for this algorithm, which was used also in QS, because it is easy o implement it and test it. Then can be replaced by another, better method. Following pseudo-algorithm was created from source code itself. So it belongs to this place.

There is a structure, with compose number property. It consists from

- n - compose number
- $digits$ - keeps count of decimal numbers in compose numbers
- $nbits$ - keeps bits length of compose number
- C - evaluate as $exp((8/9)^{(1/3)} * (ln(n))^{(1/3)} * (ln(ln(n)))^{(2/3)})$
- t - evaluate as $exp(1/2*(d*(ln(d))+sqrt((d*ln(d))^2+4*(ln(n^{(1/d)}))*ln(ln(n^{(1/d)})))))$

where d is degree of polynomial, which have to be between 3 and 10. d is set to 3 by default.

Polynomial selection

The polynomial $f(x)$ has properties:

- is irreducible over $\mathbb{Z}[x]$
- has root m modulo n

First section is selecting a good polynomial.

1. Evaluate m as $\sqrt[d]{n}$ and choose it as base of good polynomial
2. foreach $i \in (0, d * d)$
   (a) choose random m based on degree (d) of polynomial
   (b) Expand polynomial to another m

   (c) check, if polynomial has Root, if so, check if polynomial is better than previous (perfection) and save base of this polynomial (m)

3. Expand polynomial to saved goodM
4. evaluate new $m = \sqrt[d]{(n)}$
5. search right polynomial, until right will be found.
   (a) Expand polynomial to current m
   (b) check, if polynomial has Root, if so, check if polynomial is better than previous (perfection) and save base of this polynomial (m)
   (c) if $2 \cdot m^3 < n$ stop searching
   (d) decrease m
6. Expand polynomial to saved good polynomial
7. Test of reducebility

isReducible - algorithm for test reducibility of polynomial

1. foreach div, which divides $n \in (1, \sqrt{first\_argument(f)})$.
   (a) if $f(div) \cdot f(-div) \cdot f(\frac{n}{div}) \cdot f(-\frac{n}{div}) == 0$, polynomial is reducible
2. if all divisors were tested, polynomial is not reducible

Perfection - test how much is good the polynomial

1. enumerate sum of $sum = (coef(f))^2$
2. evaluate $result = \sqrt{sum/m}$
3. foreach $primes < 100$
   (a) count all numbers x, in which $f(x)$ is divisible by prime
4. evaluate $result- = count/72$, where 72 is max possible roots modulo primes up to 100
5. return $result$;

Factor bases

All numbers in factor base are primes, which application takes from file.

- Rational base is pre-evaluated as remains which are created as m, modulo primes, where m is from polynomial. Size of rational base is t from composeNum structure.
- Algebraic base is evaluated as all primes, which can divide numbers evaluated from polynomial function for $x = (0, t)$. t is from composeNum structure. There is also evaluated natural log of primes in AFB. It's size is d*t, which are taken from composeNum structure.

- Quadratic base is evaluated as all primes, which are bigger than primes in AFB or RFB, which can divide numbers evaluated from polynomial function for $x = (0, t)$. t is from composeNum structure. The size of quadratic base is made as count of decimal digits in compose numbers, which is keep in composeNum structure.

Sieving

Input are RFB , AFB , QCB ,polynomial $f(x)$, root m of $f(x)$ $(mod\, n)$, integer C

1. $b = 0$
2. $rels = []$
3. while $\#rels < \#RFB + \#AFB + \#QCB + 1$
   (a) $b = b + 1$
   (b) throw away numbers not in form $gcd(C - i, b) == 1$
   (c) evaluate norm $norm[i] = 0.5 - logabs(i) + b \cdot m)$ to get negative minimum, for $i \in [0; 2 * C]$
   (d) foreach $(p, r) \in RFB$
      i. evaluate $a = (int)(p*(-floor(C/p)) - (b*logp)) + C$, where p is prime from RFB and $logp$ is pre-evaluated logarithm of this prime
      ii. if $(a < 0)$ add p until $a >= 0$
      iii. $norm[a] += logp$ and $a += p$
      iv. then there are evaluated primes $p^2, p^3, \ldots$ same way like above.
   (e) reinitialize numbers, which was good for previous RFB - smoothness $norm[i] = fb.getLogAFB(u - 1) - logabs(algNorm(i - C, b))$ for $i \in [0; 2 * C]$, where algNorm (algebraic norm) is evaluated as $b^{deg(f)} f(\frac{C-i}{b})$
   (f) foreach $(p, r) \in AFB$
      i. evaluate $a = (int)(p * (-floor(C/p)) - (b * logp)) + C$, where p is number from AFB and $logp$ is pre-evaluated value of p
      ii. if $(a < 0)$ add p until $a >= 0$
      iii. $norm[a] += logp$ and $a += p$
   (g) for $i \in [0; 2 * C]$
      i. if $norm[i] > 0$ add $(C - i, b)$ to $rels$, for $i \in [0; 2 * C]$
4. return $rels$.

For effective using of this process, in the implementation were used pre-evaluated logarithmic values from factor base.

Linear algebra

Legendre algorithm is useful for testing numbers by primes
Input is a and p

1. if $(a\%p) == 0$, the 0 is returned.

2. Algorithm tries all numbers $< 0, p)$, if there is a number complies equation $((x * x - a)\%p) == 0$, then returns 1.

3. otherwise return -1

This part consists from two parts:

- Building the matrix
  On input of this algorithm RFB, AFB, QCB ,polynomial $f(x)$, root m of $f(x)$, list $rels = (a_0, b_0), ..., (a_t, b_t)$ of smooth pairs $(\#rels) \times (\#RFB + \#AFB + \#QCB + 1)$

  1. set all entries in $\mathbb{M}[i, j] = 0$
  2. foreach $(a_i, b_i) \in rels$
     (a) if $a_i + b_i m < 0$ set $\mathbb{M}[i, 0] = 1$
     (b) foreach $(p_k, r_k) \in RFB$
        i. let l be the biggest power of p_k that divides $a_i + b_i m$
        ii. if l is odd set $\mathbb{M}[i, 1 + k] = 1$
     (c) foreach $(p_k, r_k) \in AFB$
        i. let l be the biggest power of p_k that divides $(-b_i)^d f(-\frac{a_i}{b_i})$
        ii. if l is odd set $\mathbb{M}[i, 1 + \#RFB + k] = 1$
     (d) foreach $(p_k, r_k) \in QCB$
        i. if the Legendre symbol $\left(\frac{a_i + b_i p_k}{r_k}\right) \neq 1$ set $\mathbb{M}[i, 1 + \#RFB + \#AFB + k] = 1$
  3. return $\mathbb{M}$.

- Gaussian Elimination
  Input is $n \times m$ matrix M

  1. set $i = 1$
  2. while $i \leq n$
     (a) find first row j from row i to n where $\mathbb{M}[j, i] = 1$, if none exists try with $i = i + 1$ until one is found.
     (b) if $j \neq i$ swap row i with row j
     (c) for $i < j < n$
        i. if $\mathbb{M}[j, i] = 1$ subtract row i from row j
     (d) set $i = i + 1$
  3. for each row $0 < j \leq n$ with a leading 1 (in column k)
  4. for $0 < i < j$
     (a) if $\mathbb{M}[i, k] = 1$ subtract row j from row i
  5. return $\mathbb{M}$.

Square root

Rational square root
Input are n,polynomial $f(x)$, root m of $f(x)$, list of smooth elements the product of which is a square $deps = (a0, b0), ..., (a_t, b_t)$ and returns integer Y

1. compute the product $S(x)$ in $\mathbb{Z}[x]/f(x)$ of the elements in $deps$
2. return $Y = \sqrt{S(m) \cdot f'(m)^2} \ (mod\ n)$

Algebraic square root
Input are n,polynomial $f(x)$, root m of $f(x)$, list of smooth elements the product of which is a square $deps = (a0, b0), ..., (at, bt)$ and returns integer χ

1. find root of polynomial
2. foreach row of matrix, choose all depends row in matrix
3. determine compatible primes based on polynomial and matrix
4. compute the product $S(x)$ in $\mathbb{Z}[x]/f(x)$ of the elements in deps
5. generate random primes, based on results of compatible primes
6. choose a compatible large prime p
7. choose random $r(x) \in \mathbb{Z}_p[x]/f(x)$ with $deg(r) = deg(f) - 1$
8. compute $R_0 + R_{1_y} = (r(x) - y)^{\frac{p^d-1}{2}} \in (\mathbb{Z}_p[x]/f(x))[y]/(y^2 - S)$, i.e. compute the $\frac{p^d-1}{2}$ power of $r(x)$ modulo $y^2 - S$.
9. if $SR_1^2 \neq 1$ goto 2 and choose other p and/or $r(x)$
10. set $k = 0$
11. set $k = k + 1$
12. compute $R_{2k} = \frac{R_k(3-SR_k^2)}{2} \ (mod\ p^{2k})$
13. if $(R_k S)^2 \neq S$ goto 10
14. compute $s(x) = \pm SR_k$.
15. return $\chi = s(m)\Delta f'(m) \ (mod\ n)$

5.5.7 General number field sieve parallel

This implementation is based on previous General number field sieve. It uses openCL on part of algorithm, named „Sieve“, because that is the most exacting task which consumes almost all time. This part looks for pairs between factor basis.

According to design of application, there are more parts which can be parallelized. It was found, that choosing an polynomial consumes constant amount of time. According to experiments with previous General number field sieve it was found out, that part of looking finding linear dependencies in matrix consumes also very small and almost constant time

based on size of all basis.

Because of openCL framework doesn't support evaluation of big numbers, there was needed to create structure for evaluation of them. This structure is based on small count of vector types called uint16. The uint16 type is a vector of 16 unsigned integer variables, each of 32-bit length. This vector can be accessed in parallel. More than 16 items of vector isn't supported by openCL. There was used unsigned integer type, because we don't need use sign. And int type was used, because there is only one bigger type - ulong (64-bit integer), which must be used for result of simple operation before carry from one uint type to higher.

On this structure was created the most effective implementation of simple algorithms. Max length of number is given by ARRSIZE in files. By default is 3 - that is $2^{16*3*size(uint)-1} = 2^{1535}$.

Follows operations above the structure:

- Load - loading to this structure is realized outside of openCL. It is the same principle like we use numbers with base UINT_MAX base. There is simple need to divide big number and keep remains in lower parts of structure (picture 5.3). In this picture % is modulo, / is dividing, UM is UINT_MAX
- Negative values - because this structure are made from unsigned int format, negative value is created same as in binary format. So, negative format has all first numbers set to 1 (in binary form). Create negative version of every part of number and add 1 (complement of two).
- Add and Sub - for evaluation the ulong16 type is used, because this type can have keep carry between parts of number. The ulong16 type is a vector of 16 unsigned integer variables, each of 64-bit length. In openCL can be two numbers added the same way as basic non-vectors numbers. Results are evaluated in parallel. But there is need to add carry to higher part of number. Carry is done by division result number by (UINT_MAX +1) value and remain is given by modulo (UINT_MAX +1) value. Remain will be saved into corresponding part of structure, which made big number.
- Multiplication - this is based on multiplication of numbers. Every parts from both numbers are multiplied and shifted by sum of position of both parts in they own numbers. Then summary from all results are made (picture 5.4). Multiplication is made only for positive values. Negative values must be converted to positive. Then the result is converted to negative, if and only if one from both numbers was negative.
- Compare - two numbers are compared part by part from MSB to LSB. When difference is found, returns 1 or -1 according to difference. If no difference is found in every part of number, the 0 is returned.
- Power - it's simple multiplication by same value in cycle.
- Abs - there is check in the operation, if the number is negative. If so, the number created of the negative value of every part is evaluated and 1 is added.
- Log - creating of natural logarithm is based on divide number by natural number and save the result. But the precision wasn't good enough, so the own implementation of openCL was used, because the precision was better and the numbers for logarithm wasn't too big.

- Save - it is same procedure as load, but reversed. It's progressive taking values from the MSB entries to LSB and multiplex it by value UINT_MAX and add previous one number.

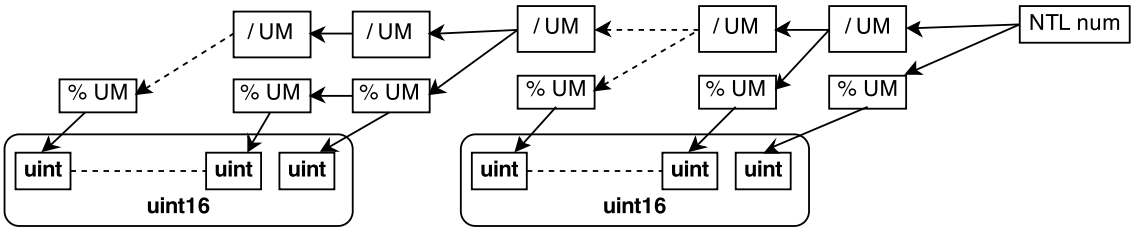


Figure 5.3: Load big NTL numbers to openCL format.

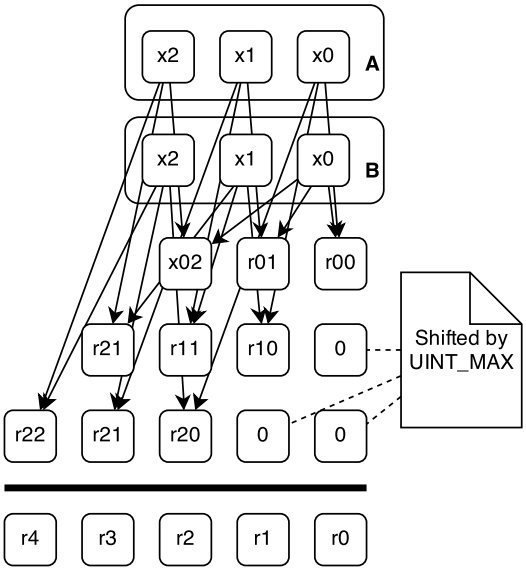


Figure 5.4: Multiplication of big numbers.

There were used global constant memory space for beginning. There weren't need any of work-group memory space, because all evaluation needs all data. So, data should be copied into another memory space, which would not improve speed of evaluation.

In test laptop was openCL headers for version 1.1, so first version of implementation is dependent on openCL 1.1.

There is also necessary to evaluate how many threads is needed for evaluation. It is processed by this system in the next order:

1. set count of threads default to 5,
2. if there is need more pairs than 250 multiple it by 2,
3. if there is need more pairs than 500 multiple it by 2 again.

Then, after first iterate of evaluation with count of threads above:

1. if remaining pairs is less than 20, add 10 threads,
2. if remaining pairs is less than 50, add 10 threads again.

5.5.8 Scripts and other auxiliary utilities

There were designed another auxiliary utilities, which provides support of factorization methods and make front-end (abstract interface) to control them. These utilities are next:

- Generator of primes - program, which uses standard NTL library to generate set of primes from 2 to entered count.
- Generator of testing numbers - program, which generate testing numbers for RSA. Generate two numbers, multiply both and if entered number is equal number of bits of result, it will be print.
- Main bash script - it controls all factoring applications. Next, it checks result and capture time to file. Afterwards, it evaluates average, mean and median for every algorithm. Application uses real part of Linux time utility. Next, it evaluate error rate, which is also saved to file. All settings of this utility are internal part of this file as global variables. Predefined bit length values for generated composite numbers are 16, 32, 64, 96 bits. These generated values are used for evaluation by all algorithms.

Chapter 6

Summary of experiments

In this chapter are specified the tests and all information gathered from algorithm are analyzed and summarized. Then will be created a result from the information. There are thoughts about breaking RSA at the end of the chapter with these or else algorithm and which length of key is safe in this time.

After some tests, there was found, there is need to slightly edit counting size of factor base (the t) to get better results and better divide the algorithm to parallel. Because when is the factor base small, most time takes part „Sieving“, and when factor base is too big, the most of time take part „Linear algebra“. This edit is $(pow(3, ((\text{Number of Bits}(n)/32) - 1))/2) * (exp(1/2 * (d * (ln(d)) + sqrt((d * ln(d))^2 + 4 * (ln(n^{(1/d)})) * ln(ln(n^{(1/d)}))))))$.

Next follows summarized statistic of all evaluations. All results are showed in tables. They were taken directly from automated script. Raw example of output is in attachments. Full outputs are on attached CD. The summarized numbers under name of algorithm is median/modus/average/standard deviation from results of all tests. Last is error rate, when the application is unable to find right result.

For running application on LINUX, you need to have installed:

- INTEL - packages based on AMD-APP-SDK-v2.9-lnx64 or packages directly from INTEL if you have right type of CPU.
- AMD - packages based on amd-catalyst-14-4-linux-x86-x86-64 and AMD-APP-SDK-v2.9-lnx64
- NVIDIA - packages based on NVIDIA-Linux-x86_64-337.19

6.1 Test on Core 2 Duo with integrated graphic card

This was evaluated on testing computer, with Core 2 duo CPU and integrated GPU. The tests have 4 parts. There were tested numbers in bit size 16, 32, 64, 96. There is need to install AMD Application SDK, because Intel does not support openCL on this type of CPU. Experiments were performed on operating system Arch-linux.

For this testing device with integrated INTEL graphic card cannot be used any Intel openCL libraries. There can be used only AMD libraries, which support INTEL by emulating evaluations on CPU instead of GPU. Because GPU is emulated by CPU, numbers are

evaluated on CPU as threads, so in result, this can be slightly worse than normal threads, because of slow CPU and transfer many of numbers to CPU. Next, openCL threads aren't dynamic. So, in result, the time of evaluation is little-bit higher. This can be caused not-optimized algorithm for big numbers, which have to be used and which doesn't have full support of openCL. This results are valid, because the time values of every used algorithms in this project were computed on same platform.

Test have fourth parts:

1. The first test was performed with 16-bit composed numbers and 90 prime numbers. Achieved results are visible in the table 6.1,
2. the second test was performed with 32-bit composed numbers and 80 prime numbers. Achieved results are visible in the table 6.2,
3. the third test was performed with 64-bit composed numbers and 40 prime numbers. Achieved results are visible in the table 6.3,
4. the fourth test was performed with 96-bit composed numbers and 45 prime numbers. Achieved results are visible in the table 6.4.

Name of algorithm	**Median**	**Modus**	**Average**	**Standard deviation**	**Error rate**
Fermat Factorization	0.0020	0.0020	0.0027	0.0037	0.00 %
Pollard's rho Algorithm	0.0030	0.0020	0.0027	0.0008	2.22 %
Pollard p-1 Factorization	0.0030	0.0030	0.0038	0.0083	1.11 %
Lenstra's Elliptic Curve Factoring Method	0.0170	0.0170	0.0176	0.0030	0.00 %
Quadratic sieve	0.0060	0.0060	0.0069	0.0020	0.00 %
Serial General number field sieve	0.3925	0.4010	0.4072	0.0660	2.22 %
Parallel General number field sieve	0.9685	0.9620	0.9889	0.0770	2.22 %

Table 6.1: Test of 16 - bit composed numbers and 90 prime numbers.

First test on testing computer was horrible. The time of computation of parallel GNFS was too high. But solution is change defined ARRSIZE, which was mentioned in implementation of GNFS (section 5.5.7) from 3 to 1. Anyway, there aren't so big numbers, so $2^{16*size(uint)-1} = 2^{511}$ is enough.

In pollard Rho and pollard P-1 can be small err-rate, because these algorithms cannot evaluate every compose number. It can be seen from definition of algorithm.

These results were taken as base for tests. There were chosen different counts of numbers, because there was needed to use this notebook for work. These tests takes a lot of time and when the tests are running, nobody can work on this notebook. So, there were

Name of algorithm	**Median**	**Modus**	**Average**	**Standard deviation**	**Error rate**
Fermat Factorization	0.0480	0.0450	0.0492	0.0063	0.00 %
Pollard's rho Algorithm	0.0030	0.0030	0.0026	0.0005	0.00 %
Pollard p-1 Factorization	0.1315	0.1100	0.1274	0.0255	0.00 %
Lenstra's Elliptic Curve Factoring Method	0.3295	0.0190	0.4198	0.3583	0.00 %
Quadratic sieve	0.2255	0.2600	0.2062	0.0757	0.00 %
Serial General number field sieve	1.3095	1.2470	1.3329	0.2119	2.50 %
Parallel General number field sieve	2.2450	2.0810	2.2725	0.2423	2.50 %

Table 6.2: Test of 32 - bit composed numbers and 80 prime numbers.

Name of algorithm	**Median**	**Modus**	**Average**	**Standard deviation**	**Error rate**
Quadratic sieve	12.1610	65.8960	28.1954	31.8887	0.00 %
Serial General number field sieve	8.7760	11.1540	9.1207	1.5781	5.00 %
Parallel General number field sieve	11.7790	11.6180	12.0739	1.9263	7.50 %

Table 6.3: Test of 64 - bit composed numbers and 40 prime numbers.

chosen different count of numbers, according to how much time algorithms spends on evaluation that number and how is this value important. There are needed numbers, which have bigger bit-length, because evaluation these numbers can be parallelized more than less bit-length and that will be useful for tests on GPU. Because 96-bit number was chosen as maximum in tests, this number have bigger count of testing numbers than previous 64-bit.

In tests with 64-bit numbers and 96-bit numbers, weren't used Fermat Factorization, Pollard's rho, Pollard p-1 and Elliptic curves, because they take very long time or didn't found right results.

This values are only the summary. On CD, in results/INTEL directory, where are full results, are some numbers, for which GNFS parallel is quicker than GNFS serial. For example for testing value 436258189671802861239807154 03 took serial GNFS 303.399 sec and parallel GNFS took only 255.688 sec for evaluation. This results are acceptable for testing on CPU. On GPU the results should be better, because it have more threads (work-units), than CPU.

Name of algorithm	Median	Modus	Average	Standard deviation	Error rate
Quadratic sieve	103.3950	53.9150	126.1310	98.5689	0.00 %
Serial General number field sieve	161.9220	150.2990	191.1550	82.1834	6.66 %
Parallel General number field sieve	182.8200	196.6090	196.6890	41.9742	8.88 %

Table 6.4: Test of 96 - bit composed numbers and 45 prime numbers.

6.2 Test on Core 2 Duo with Radeon R9 290X

Preparation of second testing computer, with Core 2 duo CPU and Radeon R9 290X, wasn't successful. These configuration were chosen:

- Fedora + catalyst 14.4 + xserver 1.14.4 + linux kernel 3.14.4 - doesn't work, fedora don't give any support to proprietal drivers.
- Arch-linux + catalyst 14.4 + (xserver 1.14, xserver 1.15.1) + linux kernel 3.14.4 - graphic card was installed, but not found for openCL purposes. Found was only CPU.

There was issue in AMD driver (fglrx module) and there was need to fix the source files of kernel-devel. There was need made it manually on fedora. In Arch-linux was better support and patches was applied, when the package was installed.

Information about available openCL can be read from program infocl, which is included in AMD-APP-SDK-v2.9-lnx64. But in this case, it shows only the CPU device, not the GPU. On the end:

- $lspci shows radeon graphic card
- xserver shows some warning about wrong port, which was good: „(WW) fglrx: No matching Device section for instance (BusID PCI:0@1:0:1) found“
- $ lsmod — grep fglrx shows loaded module
- $fglrxinfo shows good frame-rate
- $ glxinfo shows unsupported direct rendering and in detailed information shows INTEL CPU as graphic.

Because glxinfo program didn't show GPU, infocl shows only CPU device, not GPU device, which is need for testing parallel GNFS.

Change system to Arch-linux was better, but still not successful. On a lot of web pages were found, this problem is global and there are a lot of people with same problems. Some of the people solved these issues, but many of them not.

With these unsuccessful results, there was need to refuse AMD/ATI GPU for testing.

6.3 Test on Core 2 Duo with NVIDIA GeForce GTX 580

Nvidia supports openCL in opencl-nvidia package, if the NVIDIA kernel module is running. During testing on NVIDIA graphic, there also were some problems too. Source code for openCL (kernel), which was run in previous tests with good results, there took too much time on GNFS parallel.

For example, result time for 32 bit numbers was about 1 minute and for 96 bit values it take more than 30 minutes. There can be seen, that algorithm takes more than 10 times of average time of evaluation with same bit-size from summary of previous tests (section 6.1). GPU should be quicker than CPU. There was need to double-check the used device in openCL, but it was set correct to NVIDIA card.

For continue of tests, where was need to use real numbers of openCL, which shouldn't be used, because it can made precision lose and also for bigger numbers aren't big enough. But evaluation with these real numbers returns right results. Also, because in some distributions like fedora, is need to use openCL 1.2, new files are ported from openCL 1.1 to openCL 1.2. Old code can be seen on CD in folder algorithms/gnfsParallelOld, new code in algorithms/gnfsParalel.

During the tests was found, is better increase number of nodes on GPU to 50 and more, because code can be run in parallel and there is only slightly time increase between 20 threads and 50 threads. For example for 32 bit testing number for 20 threads was time 55 sec. When were used 50 threads, it was 60 sec. If there were used two evaluation in openCL time with 20 threads in both, time would be 90 sec.

For these tests was chosen count of composite numbers to 100 samples for all bit-sizes, that the results would be statistically significant with respect to evaluating by statistical functions, as compromise with respect to computing speed and relevance of the results.

For all others algorithms except GNFS Parallel can be seen similar information like on previous test (section 6.1). In elliptic curves can be improved this err-rate by set number of tests to higher number or generated factor to higher value, because this algorithm generate random elliptic curve and tries to find value in constant count of steps.

Test have five parts:

1. The first test was performed with 16-bit composed numbers and 100 prime numbers. Achieved results are visible in the table 6.5,
2. the second test was performed with 32-bit composed numbers and 100 prime numbers. Achieved results are visible in the table 6.6,
3. the third test was performed with 48-bit composed numbers and 100 prime numbers. Achieved results are visible in the table 6.7,
4. the fourth test was performed with 64-bit composed numbers and 100 prime numbers. Achieved results are visible in the table 6.8,
5. the fifth test was performed with 96-bit composed numbers and 100 prime numbers. Achieved results are visible in the table 6.9.

Name of algorithm	**Median**	**Modus**	**Average**	**Standard deviation**	**Error rate**
Fermat Factorization	0.0010	0.0010	0.0012	0.0004	0.00 %
Pollard's rho Algorithm	0.0010	0.0010	0.0010	0.0002	2.00 %
Pollard p-1 Factorization	0.0010	0.0010	0.0017	0.0058	1.00 %
Lenstra's Elliptic Curve Factoring Method	0.0100	0.0100	0.0107	0.0013	0.00 %
Quadratic sieve	0.0040	0.0040	0.0043	0.0008	0.00 %
Serial General number field sieve	0.2880	0.2790	0.2984	0.0446	1.00 %
Parallel General number field sieve	0.8725	0.8630	0.8843	0.0537	1.00 %

Table 6.5: Test of 16 - bit composed numbers and 100 prime numbers on NVIDIA.

Name of algorithm	**Median**	**Modus**	**Average**	**Standard deviation**	**Error rate**
Fermat Factorization	0.0335	0.0330	0.0348	0.0045	0.00 %
Pollard's rho Algorithm	0.0020	0.0020	0.0017	0.0004	0.00 %
Pollard p-1 Factorization	0.1015	0.1100	0.0969	0.01850	0.00 %
Lenstra's Elliptic Curve Factoring Method	0.1845	0.0730	0.3190	0.3369	0.00 %
Quadratic sieve	0.1675	0.2060	0.1568	0.0606	0.00 %
Serial General number field sieve	0.9710	0.8000	0.9898	0.1555	2.00 %
Parallel General number field sieve	2.1565	2.1270	2.1860	0.1817	2.00 %

Table 6.6: Test of 32 - bit composed numbers and 100 prime numbers on NVIDIA.

In tests with 64-bit numbers and 96-bit numbers, weren't used Fermat Factorization, Pollard's rho, Pollard p-1 and Elliptic curves, because they take very long time or didn't found right results.

From these results can be seen, GNFS parallel algorithm weren't quicker on NVIDIA GPU than GNFS serial. The algorithm was modified many ways, but it wasn't quicker. But if the evaluation of basis has been changed to smaller basis in both serial and parallel algorithm, the parallel algorithm would be quicker than serial. But this way also increases the error rate.

This values are only the summary. Full results are on CD, in path: results/NVIDIA directory.

Name of algorithm	**Median**	**Modus**	**Average**	**Standard deviation**	**Error rate**
Fermat Factorization	8.8770	7.5530	8.8895	1.2841	0.00 %
Pollard's rho Algorithm	0.0055	0.0050	0.0058	0.0022	0.00 %
Pollard p-1 Factorization	35.4845	23.5850	34.9618	6.4479	0.00 %
Lenstra's Elliptic Curve Factoring Method	5.4315	5.4190	5.3100	0.6825	96.00 %
Quadratic sieve	112.6190	108.7550	106.7490	54.1539	0.00 %
Serial General number field sieve	1.8975	1.8010	1.9977	0.3360	5.00 %
Parallel General number field sieve	4.5165	4.4600	4.6410	0.3651	5.00 %

Table 6.7: Test of 48 - bit composed numbers and 100 prime numbers on NVIDIA.

Name of algorithm	**Median**	**Modus**	**Average**	**Standard deviation**	**Error rate**
Quadratic sieve	5.1955	4.4290	19.0237	31.8222	0.00 %
Serial General number field sieve	7.6485	5.7510	9.5976	3.8220	4.00 %
Parallel General number field sieve	14.9660	11.8320	15.8022	4.4418	4.00 %

Table 6.8: Test of 64 - bit composed numbers and 100 prime numbers on NVIDIA.

Name of algorithm	**Median**	**Modus**	**Average**	**Standard deviation**	**Error rate**
Quadratic sieve	75.6820	199.9210	94.3535	70.8709	0.00 %
Serial General number field sieve	120.3200	217.2580	132.8210	41.8673	5.00 %
Parallel General number field sieve	164.1260	140.0200	184.7470	60.8516	6.00 %

Table 6.9: Test of 96 - bit composed numbers and 100 prime numbers on NVIDIA.

6.4 Results

From results can be seen, that GNFS algorithm is not a perfect, GNFS algorithm is really complicated. This can be caused by difficult last part - „square root“, which wasn't good debugged. Also, there are made some tests with evaluation of count of elements in basis. These tests proved, that size of basis can change, if the factor will be found or not.

According to results, the best algorithm for 16 and 32 bit numbers are „pollardRho“, „fermat factorization“ and „pollard p-1“ algorithms. But these algorithms cannot be used for numbers, which are equal or bigger than 64. In pollard Rho and pollard P-1 can be small err-rate, because these algorithms cannot evaluate every compose number. It can be seen from definition of algorithm.

Also can be seen, QS have a problem with primes 48-bit length, because the its result is the worst from all. Also, the QS algorithm is the most unstable algorithm. This information is clear from statistic information „standard deviation“, which value has QS the highest. This can be caused by sieving part of algorithm, which looks for smooth primes, until found enough count of them.

On GNFS serial algorithm and GNFS parallel can be also seen same dispersion. But this value is smaller because of another type of evaluation. In first test (section 6.1) are these values smaller in parallel GNFS, because of system of counting threads, which count more values than needs.

For numbers, which are bigger or equal than 64, is best algorithm serial GNFS, but implemented QS is more exact, because QS is more simpler algorithm. Parallel GNFS can be quicker algorithm, but there wasn't luck with running this algorithm on GPU of discrete graphic card.

6.5 Breaking RSA

RSA is asymmetric crypt algorithm, which can be broken only by breaking the public/private key of algorithm. This key is composed from two prime numbers. So the problem can be solved by factoring algorithm. All testing results, created by all factoring algorithms presented in this work, especially graphs shows, that factorization problem can be solved only with bigger performance of computer. But only if there won't be any true Quantum computer, which can by Principle of superposition possibly solve this problem quicker.

But even with accelerated algorithm, the result can't be taken more quicker than by using serial algorithm. Also results didn't count with distributed solving by supercomputers or large amount of computers distributed on net, which can solve results independently. There weren't even used extra performance computer. So exact length of key can't be specified. But because computers are still quicker, the key of RSA have to be larger and larger.

Following graph on picture 6.1 shows average time of test on NVIDIA graphic card from section 6.3 on bit-length. From this function can be approximately seen, how quick the time rise and evaluate time for bigger numbers. It can be seen, that function is exponential and rise too fast. According to this, the algorithm can't sure break larger values than 256-bits,

which surely would consume higher time than month. In this case, there would be better chose the QS algorithm for breaking higher bit-length numbers. But there QS algorithm wasn't chosen for parallelization, because GNFS should be quicker according to theoretical parts.

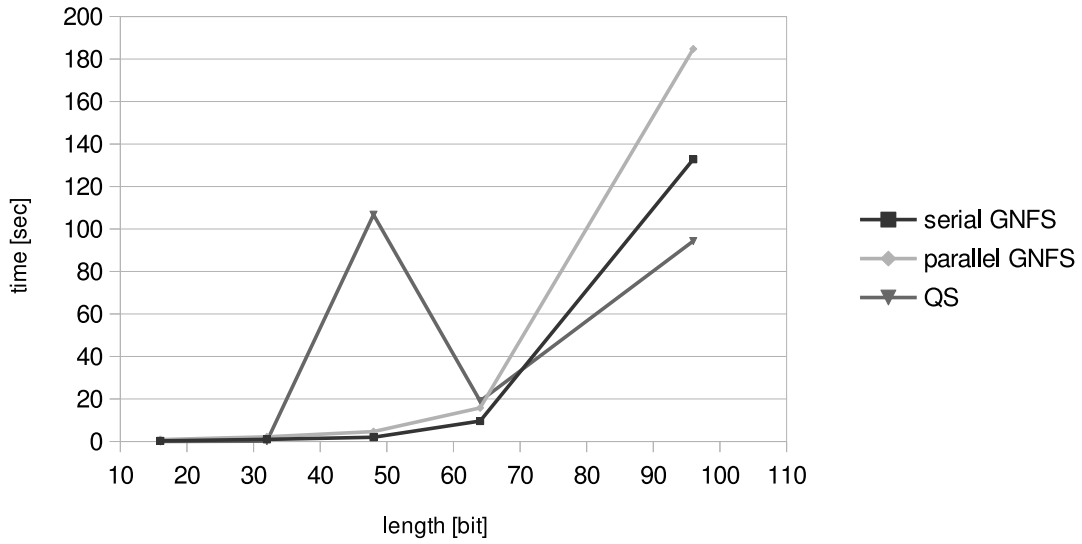


Figure 6.1: Time dependency on bit-length of factorized number.

Chapter 7

Conclusion

This chapter contains assessment of all results from algorithms, implementation of algorithms with my own work.

I begins this work by studying all famous algorithms, which can be used for factorization of integer numbers. From all the algorithms I wrote about, I like the simpler algorithms, which were easier to understand. When I saw QS algorithm first time, I didn't understand it. But after reading some books and articles, especially examples of evaluation, I understand it and I like it too. But there was quicker algorithm, which should be easier to parallelize, but which was truly difficult to understand. This algorithm is GNFS. According to my task, I have to find an algorithm, which could be parallelized the most. So, I decided to choose GNFS.

According to task of my work, I implemented serial versions of others algorithms too. The GNFS algorithm was certainly the most difficult algorithm to implement. QS algorithm was difficult, but not as much as GNFS. Then the abstract interface for testing was need to implement. Because I want this project to work on LINUX, I chose bash, which is on every LINUX distribution.

In part of parallelism, GNFS wasn't so difficult to parallelize. Not as difficult as implement serial version of GNFS. I should tried to parallelize it more. But after I tested it on real NVIDIA GPU and seen the results, there wasn't place for parallel, because other blocks have to use bigger numbers too. In result, the algorithm would be more slower.

I experimented with all algorithms, especially with GNFS in parallel form, on three graphic cards, respectively on three platforms or devices. But I couldn't achieve acceleration of evaluation with parallelization on any of those devices.

GNFS is more complicated algorithm than I expected, when I chose them in design chapter, so I couldn't assembly algorithm, which works without errors and which would be quicker in parallel by using openCL on GPU. But GNFS is still faster for bigger numbers than QS, which is much simpler.

This project can continue by distributing algorithms to more computers, which will be connected only by net, because this method are used in this time to broke all security algorithms. Then the key can be specified more precisely than in this project. For bigger

numbers can changed to parallel also part linear algebra, because bigger numbers seems to need accelerate this part. I would like to do some similar improvements on QS algorithm, which I like more than GNFS, because all it's details are easier to understand. And then perform similar test on parallel QS algorithm. That's why the QS algorithm is implemented better than GNFS algorithm.

The most confusing thing I have seen during the project was, when I saw results for parallel GNFS on NVIDIA GPU. How can be openCL so different on devices? On INTEL CPU some results were better than on serial GNFS. I expected, on GPU will be similar or better results. But I must admit, results from INTEL CPU tests were my best. I couldn't achieve so good performance on GPU.

This project gives me knowledge how can be possibility of parallelism used in algorithm„ how can be parallelism implemented, especially how can be implemented by using openCL framework. Also I learn a lot of things about factorization and which algorithms exist for it's solving.

Bibliography

[1] A Oliver L Atkin and François Morain. Elliptic curves and primality proving. *Mathematics of computation*, 61(203):29–68, 1993.

[2] David M Bressoud. Factorization and primality testing. *New York*, 1989.

[3] Matthew E Briggs. *An introduction to the general number field sieve.* PhD thesis, Virginia Polytechnic Institute and State University, 1998.

[4] H. Cohen. *A course in computational algebraic number theory.* Springer, 1993.

[5] C Crandall, R.; Pomerance. *Prime numbers. A computational perspective.* Springer, 2nd edition, 2005.

[6] Rob Farber. Opencl^{TM} – portable parallelism part2, part3. `www.codeproject.com/Articles/110685/Part-1-OpenCL-Portable-Parallelism`, 2010.

[7] Joseph L Gerver. Factoring large numbers with a quadratic sieve. *Mathematics of Computation*, 41(163):287–294, 1983.

[8] Brent Oster Greg Ruetsch. Getting started with cuda, 2008.

[9] Ramanujachary Kumanduri and Cristina Romero. *Number Theory with Computer Applications.* Prentice hall, Upper Saddle River, 1998.

[10] Arjen K Lenstra, Hendrik W Lenstra Jr, Mark S Manasse, and John M Pollard. The number field sieve. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 564–572. ACM, 1990.

[11] H. W. Jr. Lenstra. *Factoring Integers with Elliptic Curves*, volume 126 of *2nd.* Annals of Mathematics, 8th revised edition, Nov. 1987.

[12] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography.* CRC press, 1996.

[13] Peter L. Montgomery. *Square roots of products of algebraic numbers. In Mathematics of Computation 1943–1993.* American Mathematical Society, 1994.

[14] Aaftab Munshi et al. The opencl specification. *Khronos OpenCL Working Group*, 1:l1–15, 2009.

[15] Phong Nguyen. *A montgomery-like square root for the number field sieve. In Proceedings of ANTS-III*, volume 1423. Springer-Verlag, 1998.

[16] Carl Pomerance. *Cryptology and Computational Number Theory; Factoring*, volume 126 of *2nd*. AMS, Providence, 8th revised edition, 1990.

[17] Carl Pomerance. A tale of two sieves. *Notices Amer. Math. Soc*, 43:1473–1485, 1996.

[18] Carl Pomerance. Smooth numbers and the quadratic sieve. In *Proc. of an MSRI workshop, J. Buhler and P. Stevenhagen, eds.(to appear)*, 2008.

[19] Hans Riesel. *Prime numbers and computer methods for factorization*, volume 126. Springer, 1994.

[20] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[21] Matthew Scarpino. A gentle introduction to opencl. `www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854`, 2011.

[22] Byunghyun Schaa, Dana; Jang. Programming with cuda and opencl. `www.ece.neu.edu/ dschaa/files/cuda_opencl_hms.pdf`, 2010.

[23] Robert D Silverman. The multiple polynomial quadratic sieve. *Mathematics of Computation*, 48(177):329–339, 1987.

[24] Peter Stevenhagen. The number field sieve. *Surveys in Algorithmic Number Theory, edited by JP Buhler and P. Stevenhagen, Math. Sci. Res. Inst. Publ*, 44:83–100, 2005.

[25] Eric W. Weisstein. Fermat, pierre de. `http://scienceworld.wolfram.com/biography/Fermat.html`, 2008-11-01 [cit. 2008-11-28].

[26] Eric W. Weisstein. Brent's factorization method. `http://mathworld.wolfram.com/BrentsFactorizationMethod.html`, December 28, 2002.

[27] Eric W. Weisstein. Pollard p-1 factorization. `http://mathworld.wolfram.com/Pollardp-1FactorizationMethod.html`, December 28, 2002.

[28] Eric W. Weisstein. Pollard rho factorization. `http://mathworld.wolfram.com/PollardRhoFactorizationMethod.html`, December 28, 2002.

Appendix A

Content of CD

Attached CD contains following folders:

- /latex-doc - contains whole documentation of Diploma work as pdf file and source files of documentation,
- /algoritms - contains whole structure of factoring algorithms and testing scripts,
  - /algoritms/elipticCurves.cpp - factoring algorithm Lenstra's Elliptic Curve,
  - /algoritms/fermat.cpp - factoring algorithm Fermat Factorization,
  - /algoritms/gnfsParallel - factoring algorithm General number field sieve in parallel form, which computes results on NVIDIA graphic card,
    - /algoritms/gnfsParallel/fBase.cpp
    - /algoritms/gnfsParallel/fBase.hpp
    - /algoritms/gnfsParallel/lAlgebra.cpp
    - /algoritms/gnfsParallel/lAlgebra.hpp
    - /algoritms/gnfsParallel/main.cpp
    - /algoritms/gnfsParallel/Makefile
    - /algoritms/gnfsParallel/sieve.cpp
    - /algoritms/gnfsParallel/sieve.hpp
    - /algoritms/gnfsParallel/sqrt.cpp
    - /algoritms/gnfsParallel/sqrt.hpp
  - /algoritms/gnfsParallelOld - factoring algorithm General number field sieve in parallel form, which computes results on integrated graphic card,
    - /algoritms/gnfsParallelOld/fBase.cpp
    - /algoritms/gnfsParallelOld/fBase.hpp
    - /algoritms/gnfsParallelOld/lAlgebra.cpp
    - /algoritms/gnfsParallelOld/lAlgebra.hpp
    - /algoritms/gnfsParallelOld/main.cpp
    - /algoritms/gnfsParallelOld/Makefile
    - /algoritms/gnfsParallelOld/sieve.cpp
    - /algoritms/gnfsParallelOld/sieve.hpp
    - /algoritms/gnfsParallelOld/sqrt.cpp

        - /algoritms/gnfsParallelOld/sqrt.hpp
    - /algoritms/gnfsSerial - factoring algorithm General number field sieve in serial form,
        - /algoritms/gnfsSerial/fBase.cpp
        - /algoritms/gnfsSerial/fBase.hpp
        - /algoritms/gnfsSerial/lAlgebra.cpp
        - /algoritms/gnfsSerial/lAlgebra.hpp
        - /algoritms/gnfsSerial/main.cpp
        - /algoritms/gnfsSerial/Makefile
        - /algoritms/gnfsSerial/sieve.cpp
        - /algoritms/gnfsSerial/sieve.hpp
        - /algoritms/gnfsSerial/sqrt.cpp
        - /algoritms/gnfsSerial/sqrt.hpp
    - /algoritms/Makefile - global make file,
    - /algoritms/primeGen.cpp - generating program for primes,
    - /algoritms/primes list of primes, generated by primeGen,
    - /algoritms/pollardP1.cpp - factoring algorithm Pollard p-1,
    - /algoritms/pollardRho.cpp - factoring algorithm Pollard's rho,
    - /algoritms/qs - factoring algorithm Quadratic sieve,
        - /algoritms/qs/main.cpp
        - /algoritms/qs/Makefile
        - /algoritms/qs/primes.txt - list of primes, generated by primeGen, which also includes -1,
        - /algoritms/qs/qs
    - /algoritms/Readme.txt - readme file contains simple description for all algorithms,
    - /algoritms/script.bash - testing script, which runs and manage all other scripts and algorithms,
    - /algoritms/testNumbers* - files with test numbers, generated by testNumbersGen. Number is bit-length of numbers, which contains,
        - /algoritms/testNumbers[16,32,48,64,96]
    - /algoritms/testNumbersGen.cpp - program, which generate test numbers,
- /results - contains results of run testing scripts on different platforms. Number is bit-length of numbers, which contains,
    - INTEL - results from integrated GPU (emulation of CPU)
        - taken output from script into file,
            + /results/INTEL/times[16,32,64,96]
        - taken output from script.bash into file. Files with times of evaluation, every row corresponds with same number row from testNumbers*,
            + /results/INTEL/fullLog[16,32,64,96]
    - NVIDIA - results from NVIDIA GPU
        - taken output from script into file,

  + /results/NVIDIA/times[16,32,48,64,96]
- taken output from script.bash into file. Files with times of evaluation, every row correspond with same number row from testNumbers*,
  + /results/NVIDIA/fullLog[16,32,48,64,96]

Appendix B

Manual

Manual for testing scripts and factoring algorithms.

You can run tests by typing „bash script.bash“. For this, you need to have awk, bash, bc installed. bc is needed, because bash can't evaluate big numbers, so it can't check if the numbers was correct.

Or you also can make programs simply by type „make“ or better „make -B“
and run every program separately this way.

every program can be run by command:

```
./program <number> [primes] [-rsa]
```

where:
`<number>` is composite number for testing.
`<primes>` is mandatory argument for creating base for algorithms:
„qs“, „gnfsSerial“, „gnfsParallel“

You need to enter the path to file of pre-generated primes.
This file can be generate by program „primeGen“.

Parameter -rsa is useful for testing, it creates only one factor from composite numbers, so it is good for RSA numbers, which consists from only two primes.

You can also change configuration of script.bash
There are few variables, which control the program.

- testNumbersBits=(16 32) # every number means testing by this numbers, which bit-length is this value. In this case factor programs will be tested by 16-bits, 32-bits values.

- testNumbersCount=(80 60) # this is pair to testNumbersBits. This value says, how much test numbers will contain the file.

- algorithms=(1 1 1 1 1 1 1) #(fermat pollardrho pollardp1 elipticcurves qs gnfsserial gnfsparallel) - enable or disable testing of some algorithm. 1 means enabled.

B.1 Example of raw output from testing script

Numbers under name of algorithm is returned result. If returned number is right result, the word OK is by this number. Because composed numbers consists from two primes, there can be two right factors.

```
TEST NUMBER 32881
FERMAT    POLLARDRHO    POLLARDP1    ELIPTIC       QS    SERGVFS    PARGVFS
OK 131        OK 131       OK 131     OK 131   OK 251     OK 251     OK 251
------------------------------------------------------------------------------
TEST NUMBER 33017
FERMAT    POLLARDRHO    POLLARDP1    ELIPTIC       QS    SERGVFS    PARGVFS
OK 137        OK 241       OK 241     OK 137   OK 137     OK 241     OK 241
------------------------------------------------------------------------------
TEST NUMBER 33043
FERMAT    POLLARDRHO    POLLARDP1    ELIPTIC       QS    SERGVFS    PARGVFS
OK 173        OK 173       OK 191     OK 173   OK 173     OK 173     OK 173
------------------------------------------------------------------------------
TEST NUMBER 33227
FERMAT    POLLARDRHO    POLLARDP1    ELIPTIC       QS    SERGVFS    PARGVFS
OK 149        OK 223    ERR 33227     OK 223   OK 149     OK 149     OK 149
------------------------------------------------------------------------------
TEST NUMBER 33389
FERMAT    POLLARDRHO    POLLARDP1    ELIPTIC       QS    SERGVFS    PARGVFS
OK 173        OK 173       OK 193     OK 193   OK 173     OK 173     OK 173
------------------------------------------------------------------------------
TEST NUMBER 33499
FERMAT    POLLARDRHO    POLLARDP1    ELIPTIC       QS    SERGVFS    PARGVFS
OK 139        OK 241       OK 241     OK 139   OK 241     OK 241     OK 241
------------------------------------------------------------------------------
TEST NUMBER 34081
FERMAT    POLLARDRHO    POLLARDP1    ELIPTIC       QS    SERGVFS    PARGVFS
OK 173        OK 197       OK 197     OK 197   OK 173     OK 173     OK 173
```

More detailed results from script are leaved on CD with the project.